

日 本 国 特 許 庁
JAPAN PATENT OFFICE

07.10.03

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office.

出 願 年 月 日 2 0 0 2 年 1 0 月 2 8 日
Date of Application:

出 願 番 号 特 願 2 0 0 2 - 3 1 3 2 0 1
Application Number:
[ST. 10/C]: [J P 2 0 0 2 - 3 1 3 2 0 1]

出 願 人 株 式 会 社 ル ネ サ ス テ ク ノ ロ ジ
Applicant(s):

REC'D 21 NOV. 2003

WIPO PCT

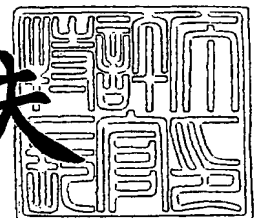
**PRIORITY
DOCUMENT**

SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH RULE 17.1(a) OR (b)

2 0 0 3 年 1 1 月 7 日

特許庁長官
Commissioner,
Japan Patent Office

今 井 康 夫



【書類名】 特許願

【整理番号】 H02015531

【提出日】 平成14年10月28日

【あて先】 特許庁長官殿

【国際特許分類】 G06F 19/00

【発明者】

【住所又は居所】 東京都小平市上水本町五丁目 2 0 番 1 号 株式会社日立
製作所 半導体グループ内

【氏名】 谷本 匡亮

【発明者】

【住所又は居所】 東京都小平市上水本町五丁目 2 0 番 1 号 株式会社日立
製作所 半導体グループ内

【氏名】 鎌田 丈良夫

【特許出願人】

【識別番号】 000005108

【氏名又は名称】 株式会社日立製作所

【代理人】

【識別番号】 100089071

【弁理士】

【氏名又は名称】 玉村 静世

【電話番号】 03-5217-3960

【手数料の表示】

【予納台帳番号】 011040

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 システム開発方法及びデータ処理システム

【特許請求の範囲】

【請求項1】 並列動作の記述が可能なプログラム言語を用いて複数のデバイスを定義したプログラム記述を入力し、入力したプログラム記述を中間表現に変換し、この中間表現に対し、実時間制約を満足するパラメータを生成し、生成したパラメータに基づいてハードウェア記述言語による回路記述を合成することを特徴とするシステム開発方法。

【請求項2】 前記中間表現は、コンカレントなコントロールフローフラグ、コンカレントなパラメータ付き時間オートマトン、又はパラメータ付き時間オートマトンであることを特徴とする請求項1記載のシステム開発方法。

【請求項3】 前記パラメータ生成に、パラメトリック・モデルチェックングを行うことを特徴とする請求項2記載のシステム開発方法。

【請求項4】 前記実時間制約はR P C T Lで与えられることを特徴とする請求項3記載のシステム開発方法。

【請求項5】 前記プログラム記述はr u nメソッドを用いてデバイスの定義を行い、バリア同期を用いてデバイスのクロック同期を定義することを特徴とする請求項4記載のシステム開発方法。

【請求項6】 並列動作の記述が可能なプログラム言語を用いて複数のデバイスを定義したプログラム記述を入力し、入力したプログラム記述を中間表現に変換し、この中間表現に対し、実時間制約を満足するパラメータを生成し、生成したパラメータに基づいてハードウェア記述言語による回路記述を合成することを特徴とするデータ処理システム。

【請求項7】 前記プログラム記述はr u nメソッドを用いてデバイスの定義を行い、バリア同期を用いてデバイスのクロック同期を定義することを特徴とする請求項6記載のデータ処理システム。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】

本発明は、並列動作を記述できる言語からデジタル回路を開発する方法、更には並列動作を記述できる言語からデジタル回路のハードウェア合成を行うデータ処理システムに関する。

【0002】

【従来の技術】

近年、モバイル・コンピューティング環境を実現する上で、システムLSIの果たす役割はその重要性を増している。また、モバイル・コンピューティングでは実時間制約を如何に満たすかが、しばしば問題となる。更に、システムLSIを実装する上で要求性能を満たすよう設計する場合、バス・システムの設計が重要となる。然るに、バス・システムを実時間制約を満たすよう効率良く設計する為の設計手法がシステム・シミュレーションによる方法以外提案されていないのが現状である。システムシミュレーションについて記載された文献の例として下記のものがある。

【0003】

【特許文献1】

特開2002-279333号公報

【特許文献2】

特開2000-035898号公報

【特許文献3】

特開平07-084832号公報

【0004】

【発明が解決しようとする課題】

本発明の目的は、Java（登録商標）等の並列動作を記述可能なプログラム言語を用いてバス・システム等のハードウェア設計の工数を低減することにある。

【0005】

本発明の目的は、並列動作を記述可能なプログラム言語とパラメトリック・モデルチェッキングを用いた、実時間制約を満たすバス・システムの新規設計手法を提供することにある。

【0 0 0 6】

本発明の別の目的は、モデルチェッキング技術とハードウェア合成技術の融合による新たな設計手法を提供することにある。

【0 0 0 7】

本発明のさらに別の目的は、実時間制約を有するバス・システムに対する並列動作記述可能なプログラム言語によるモデル化とパラメトリック・モデルチェッキングによる検証、更にはハードウェア合成を実現することにある。

【0 0 0 8】

本発明の前記並びにその他の目的と新規な特徴は本明細書の記述及び添付図面から明らかになるであろう。

【0 0 0 9】

【課題を解決するための手段】

本願において開示される発明のうち代表的なものの概要を簡単に説明すれば下記の通りである。

【0 0 1 0】

すなわち、並列動作の記述が可能なプログラム言語を用いて複数のデバイスを定義したプログラム記述を入力し、入力したプログラム記述を中間表現に変換し、この中間表現に対し、実時間制約を満足するパラメータを生成し、生成したパラメータに基づいてハードウェア記述言語による回路記述を合成する。

【0 0 1 1】

上記より、J a v a (登録商標)等の並列動作を記述可能なプログラム言語によりバスシステム等に対するモデル化を行って、実時間制約を満たすシステムの設計を行うことができる。これにより、ハードウェア設計の工数低減が可能になる。

【0 0 1 2】

本発明の一つの形態として、前記中間表現は、コンカレントなコントロールフローグラフ、コンカレントなパラメータ付き時間オートマトン、又はパラメータ付き時間オートマトンである。

【0 0 1 3】

本発明の一つの形態として、前記パラメータ生成に、パラメトリック・モデルチェックングを行う。モデルチェックング技術とハードウェア合成技術の融合による新たな設計手法を提供することができる。

【0014】

本発明の一つの形態として、前記実時間制約はR P C T Lで与えられる。

【0015】

本発明の一つの形態として、前記プログラム記述はr u nメソッドを用いてデバイスの定義を行い、バリア同期を用いてデバイスのクロック同期を定義する。

【0016】

【発明の実施の形態】

本発明の一例として、実時間制約を有するバス・システムのJ a v a（登録商標）によるモデル化及びパラメトリック・モデルチェックングによる検証とハードウェア合成について説明する。本明細書ではJ a v a（登録商標）を単にジャバ言語とも記す。

【0017】

《設計方法の全体》

図1には設計方法の全体が示される。設計対象とされるシステムはジャバ言語による記述（ジャバ言語記述）でモデル化される（S1）。ジャバ言語記述によるモデル化では、単一バス上のデバイスをジャバ言語のrun()メソッドを用いて記述する。run()メソッドはマルチスレッドを構成するスレッドにおいて実行させたいプログラムコードがその()内に記述される。クロックはバリア同期を用いて実現される。一般的にバリア同期とは複数のモジュールからのデータを受信する場合に同時刻に処理されるべき全てのデータを待つための同期手法として把握することができる。

【0018】

次いで、S1で生成されたジャバ言語記述（ジャバコード）1を読み込み、中間表現に変換する（S2）。ここで、中間表現2は、コンカレントなコントロール・フロー・グラフ（以下、C-C F G）、コンカレントなパラメータ付き時間オートマトン（以下、C-T N F A）、パラメータ付き時間オートマトン（以下、

TNFA)からなる。CFG (コントロール・フロー・グラフ) は一般に関数内部において制御の流れを示すグラムを意味する。TNFA (オートマトン) とは、有限の種類の入力を離散的な時刻に受け、過去から現在までに入力された入力の系列を、回路で決まった個数以下のクラスに分類して記憶し、それに基づいて有限の種類 of 出力を出す回路の論理的なモデルとして把握することができる。

【0019】

中間表現2として得られたTNFAとRPCTL (Real Time Parametric Computation Tree Logic) 3で記述された実時間制約を読み込み、パラメトリック・モデル・チェックを実行し(S3)、入力RPCTL等を満たすパラメータ条件4を導出する。

【0020】

満足するパラメータ条件がなければ方式変更を行いジャバ言語記述を修正し、満足するパラメータ条件があれば、そのパラメータ条件をサイクル制約として、C-CFGを読み込んで高位合成(S4)によりHDL (hardware Description Language) による回路記述5へ変換する。回路記述は例えばRTL (Register Transfer Level) である。

【0021】

上記開発方法はそれを実現するためのプログラムをコンピュータ装置で実行することによって行なわれる。ジャバ言語記述からHDLを得る為の開発支援プログラムはデジタル回路の設計支援プログラムとして位置付けることができる。

【0022】

本設計手法により、単一バスシステムの上流工程でのモデル化、検証が可能となり、かつ実時間制約を満たすハードウェアの設計自動化が可能となる。上流工程での検証が可能なのは、設計対象システムはジャバ言語記述でモデル化され、その記述自身が実行可能にされるからである。

【0023】

《ジャバ言語によるモデル化に対する制約》

ジャバ言語によるモデル化に際してのジャバ言語記述とバスシステムに対する制約について説明する。単一バス・システムのジャバ言語によるモデル化を目的

とするため、ジャバ言語記述に対し、

- 1) ダイナミック・インスタンス化の禁止、
- 2) run()メソッドからのstart()メソッドコールの禁止、

の制約を置く。制約1)はLSIのプロセス変更に関するものであり、ここではハードウェアを扱っていることから、許容可能な制約であると考えられる。また、制約2)はバス・プロトコルの検証を目的とする限りに於いては、直接バス動作に関わる部分のみをモデル化すれば良い為、許容可能な制約であると考えられる。

【0024】

また、パラメトリック・モデルチェッキング・ツールの制約から、モデルに対して、

- 3) 単一双方向バス・システム、
- 4) バス権制御は固定優先度
- 5) バス上の各デバイスは一定周期で処理を終了、の制約を課す。上記制約の緩和は新たな課題として位置付けられる。

【0025】

《ジャバ言語によるモデル化の為のデザインパターン》

図2には単一バス・システムのジャバ言語によるモデル化の為のデザインパターンが例示される。単一バス・システムのジャバ言語によるモデル化はUML (Unified Modeling Language) で与えられる図2のデザインパターンに沿って記述される。バス上の各デバイス動作をDeviceImpl Class内のrun()メソッドに実装し、バスを介してアクセスがあるデバイス内レジスタはRegister Class内のattribute (属性) として実装し、バスを介しての同期通信メソッドをRegister Classのメソッドとして実装する。また、バスに対応するBus Class、バスのロック管理を行うBusController Class、及びクロック同期を管理するClock Controller Classを実装する。バスのロック管理とはバスアービトレーションの制御を意味する。図2の標記において、棒線を付した三角記号△は継承を意味し、例えばDeviceImpl ClassはDevice Classの子クラスであり、DeviceImpl ClassはDevice Classのメソッドを使用可能である。記号→はユース (use) を意味し、例えばD

evice ClassはClock Controller Classのメソッドを利用する。

【0026】

尚、ジャバ言語コードの再利用性を高めるため、下記方針

- 1) デバイスの追加・削除 (DeviceImpl Classの追加・削除)、
- 2) デバイス動作の変更 (DeviceImpl Classのrun()メソッドの変更)、
- 3) 共有変数の追加・削除 (Register Classのattributeの追加・削除)、
- 4) バスプロトコルの追加・変更 (Register Classの同期通信メソッドの追加・変更)、

にて、ジャバ言語コードの変更が可能となるようデザイン・パターンを定義している。

【0027】

図2のデザインパターンにおける各クラス (Class) について説明する。

【0028】

(1) Deviceクラス

デバイスに共通の要素をまとめた抽象クラスである。レジスタやバスは複数ある場合でも処理内容自体が異なることはないため、同じクラスオブジェクトを複数生成ことにより複数存在することを表現できるが、デバイスはデバイスごとにその処理内容が異なる。よって、並列動作を行なうためにThreadクラスを継承し、Clock controller Classのインスタンス、Register Classのインスタンス、Bus Classのインスタンスを表すメンバ変数、各デバイスがアクセス可能な共有レジスタの情報を登録する為のメンバ変数、及び各デバイスがアクセス可能な共有レジスタの情報を登録する為のメソッドといったようなデバイスに共通な要素をまとめたクラスとしてDeviceクラスを実装する。

【0029】

(2) DeviceImplクラス

実際のデバイスにあたるクラスである。デバイスはデバイスごとに処理内容が異なるため、デバイスに必要な要素をまとめたDeviceクラスを継承し、その処理内容を記述する。つまり、Aという処理を行なうDeviceImplAクラス、Bという処理を行なうDeviceImplBクラスといったように処理ごとにDeviceクラスの実装ク

ラスが存在することになる。

【0030】

(3) Registerクラス

共有レジスタに対応するクラスである。レジスタの値を表すメンバ変数と、その値をリード/ライトするメソッドから成る。

【0031】

(4) Busクラス

バスに対応するクラスである。バスが使用中であるか否かの状態を表すメンバ変数、そのバスに繋がっている共有レジスタをあらわすメンバ変数、及び、状態を変更するメソッドから成る。

【0032】

(5) BusControllerクラス

バスを介した共有レジスタへのアクセス時のバスロックとバスのロック解放を行なうクラスである。バスのロック・ロック解放はこのクラスに対して依頼する。特に、バスに対して1つデバイスがアクセスを行うとフラグ変数を1に設定するメソッドを含む。このフラグをバスアクセス・フラグとしてメンバ変数として持ち、このメンバ変数により、同一クロック内での複数回のバスロック動作を回避している。

【0033】

(6) ClockControllerクラス

クロック管理クラスである。バスシステム上の各デバイスにクロック同期動作を行なわせるために、1クロック分の処理終了の通知を集め、全デバイスの処理終了が認められたら、次のクロックの処理を行なってよいという通知を行うと同時にBusController Classのバスアクセス・フラグを0にリセットする。

【0034】

《モデル化におけるクロック同期メカニズム》

クロック同期メカニズムについて説明する。図3に示すバリア同期によるクロック同期メソッドを用いて、クロックを実現する。具体的には、1クロック分の処理終了の通知を集め、全デバイスの処理終了が認められたら、次のクロックの

処理を行なってよいという通知を行う。

【0035】

また、バスのロックが残っていない場合には、Bus Classのバスアクセス・フラグを0にリセットする。尚、クロック遷移はパラメトリック・モデル・チェックへの入力に際して、パラメータ化される為、必ずしも1クロック単位であるとは限らない。

【0036】

このクロック遷移のパラメータ化により、サイクル精度でのモデル化よりも抽象度の高いモデル化を実現する。

【0037】

レジスタへの値書き込みには1クロックを要する為、それを実現する必要がある。これは、consume_1_clockメソッド内で、図4に例示されるRegister Classの assignWriteValue メソッドをコールする事で実現している。

【0038】

《モデル化におけるバス権獲得メカニズム》

バス権管理の同期メカニズムとして、先ずバス権獲得メカニズムについて説明する。バス権獲得メカニズムは、図5に示すメソッドにより、バス権獲得を管理する。バス権の要求はgetBusLockメソッドにより行う。図6には図5の各メソッドの呼び出し関係を表すコールグラフが示される。図7及び図8には上記バス権獲得メカニズムのジャバ言語記述が例示される。

【0039】

《モデル化におけるバス権解放メカニズム》

次にバス権解放メカニズムについて説明する。バス権解放メカニズムは図9に示すメソッドにより、バス権解放を管理する。バス権の解放はfreeBusLockメソッドにより行う。図10には図9の各メソッドの呼び出し関係を表すコールグラフが示される。図11には上記バス権解放メカニズムのジャバ言語記述が例示される。

【0040】

《モデル化における排他的同期リード・メソッド》

バスへの排他的同期アクセス方式として、排他的同期リード・メソッドについて説明する。排他的同期リード・メソッドには以下のシングルリードとバーストリードがある。シングルリードはsync_read メソッドをrun()でコールする事で実現される。バーストリードは、sync_burst_read、endBurstAccess、consume_clock メソッドをrun()の、synchronizedブロック内で用いる事で実現される。

【0041】

図12にはsync_readメソッドのジャバ言語コード、図13にはrun()でのsync_readメソッド記述例が示される。run()での記述例に現れる未定義メソッド、即ち、do_something_w_or_wo_clock_boundary()はクロック境界を含んでもよい何らかの処理を意味し、do_something_wo_clock_boundary()はクロック境界を含まない何らかの処理を意味する。

【0042】

図14にはsync_burst_readのジャバ言語コード、図15にはendBurstAccessのジャバ言語コード、図16にはfreeBurstBusLockのジャバ言語コード、図17にはrun()でのバーストリードの記述例が示される。バーストリードでは、バスのロックはコールする度に重ねて獲得し、バスの解放を行わないで値を返す。そして、バーストリード回数のリードが終了すると、重ねたロックを一気に解放するという実装方法となっている。この実装により、毎サイクル、リード値が返ってくるというバースト動作を実現している。

【0043】

《モデル化における排他的同期ライト・メソッド》

バスへの排他的同期アクセス方式として、排他的同期ライト・メソッドについて説明する。排他的同期ライト・メソッドには以下のシングルリードとバーストリードがある。シングルライトはsync_write メソッドをrun()でコールする事で実現される。バーストライトはsync_burst_write、endBurstAccess、consume_clock メソッドをrun()の、synchronizedブロック内で用いる事で実現される。

【0044】

図18にはsync_writeのジャバ言語コードが示され、図19にはrun()でのsync_writeメソッド記述例が示される。

【0045】

図20にはsync_burst_writeのジャバ言語コードが示され、図21にはrun()でのバーストライトの記述例が示される。バーストライトでは、バスのロックはコールする度に重ねて獲得し、書き込み処理終了後バスの解放を行わないで処理を終了する。そして、バーストライト回数のライトが終了すると、重ねたロックを一気に解放するという実装方法となっている。この実装により、毎サイクル、ライト値が書き込めるというバースト動作を実現している。

【0046】

《ジャバ言語記述による実装例》

ジャバ言語記述による実装例を説明する。図22には実装例の概略仕様が示される。ここでは、共有メモリ方式の2次元グラフィックス描画・表示システムを一例とする。同図に示されるシステムは、コマンドインタフェース (Command Interface)、ユニファイドメモリ (Unified Memory)、グラフィック描画ユニット (Graphics Rendering Unit)、及び表示ユニット (Display Unit) を有し、それらは双方向バス (Single bi-direction Bus) に共有する。

【0047】

(1) Command Interfaceは、外部からの描画コマンドを受付け、バスを介して受け付けた描画コマンドをメモリへ転送する。

【0048】

(2) Unified Memoryは、描画コマンド、描画ソースデータ、表示データを一元的に格納するメモリである。システムの低コスト化・部品点数削減に効果大。モデルを単純化する為、Javaでは配列で表現する。本デバイスはスレーブデバイスである。

【0049】

(3) Graphics Rendering Unitは、Unified Memoryに描画コマンドがあるかをポーリングで調べ、存在する場合は、描画コマンド、描画データをバスを介してリードしながら描画処理を行い、描画結果を内部バッファに格納し、一気に Unified Memoryへバースト転送する。モデルを単純化する為、描画コマンド、描画データ転送は配列への連続アクセスで実現する。

【0050】

(4) Display Unitは、表示データをリードしながら1ラインずつ表示を行う。モデルを単純化する為、垂直同期はモデル化しない。また、水平帰線期間はバスアクセスを行わないものとする。

【0051】

上記コマンドインタフェース (Command Interface)、グラフィック描画ユニット (Graphics Rendering Unit)、及び表示ユニット (Display Unit) がバスマスタデバイスであり、ユニファイドメモリ (Unified Memory) がバススレーブデバイスである。バスマスタデバイスが同時にバス権獲得を行った場合の、バス権獲得の優先順位は、表示ユニット (Display Unit) > コマンドインタフェース (Command Interface) > グラフィック描画ユニット (Graphics Rendering Unit) とする。

【0052】

次に、図22の仕様に対するrun() methodの実装例について説明する。

【0053】

(1) Command Interface

まず、コマンドインタフェースに関するrun() methodの実装例を説明する。その実装例は図23乃至図26に例示される。外部入力write_req信号（具体的にはCPUからのライト信号）があり且つwait出力信号の値決定に用いる内部変数wait_flagがfalseだとコマンド受付を実行（一定サイクル消費）し、その後にはwait_flag変数をtrueとする。次いで、バス権取得を試み、取得できたらUnified Memoryのコマンドフラグ0, 1をsync_burst_readで読み出し、値が0（描画コマンド処理済み）であるコマンドフラグに対応するメモリ領域に対して、バス権を取得した状態のままで、sync_burst_writeを実行し、コマンドフラグ、描画コマンド、描画ソースデータを転送し、wait_flag変数をfalseとする。但し、モデル簡略化の為、例えばコマンドフラグが両方とも0であっても、コマンドフラグ0に対応するメモリ領域へのsync_burst_writeによる書き込みのみを実行するものとし、描画ソースデータの転送は省略する。

【0054】

write_flag変数がtrueで、write_req入力信号がtrueの場合のみ、wait信号をtrueとし、それ以外はwait信号をfalseとする。特に、コマンド受付実行中はwait_flag変数をfalseとする。尚、wait信号、wait_flag変数ともに初期値はfalseとする。

【0055】

また、Command Interfaceが受け付けるコマンドは、コマンドフラグ、描画コマンド、描画ソースデータ、テクスチャデータの格納開始・終了アドレスからなっており、テクスチャデータの格納開始・終了アドレスは描画コマンドに含まれているものとする。

【0056】

(2) Unified Memory

ユニファイドメモリに関するrun() methodの実装例を説明する。その実装例は図27に例示される。グラフィックス描画コマンド、描画ソース・グラフィックス・データ、描画後の表示用データを一元的に格納しているメモリ。モデル化を簡略化する為、何も実行しないスレッドとして実現。このメモリの実体はRegister Classのインスタンスとしてrun()メソッド内でアローケートされている。

【0057】

尚、モデル簡略化の為、メモリ配列を下記

mem_con_reg.current_value[0]: コマンドフラグ0、
mem_con_reg.current_value[1]: 描画コマンド0、
mem_con_reg.current_value[2]: コマンドフラグ1、
mem_con_reg.current_value[3]: 描画コマンド1、
mem_con_reg.current_value[4]: 描画ソース・データ、及び描画後の表示用データ、の構造としている。

【0058】

実際には、描画ソース・データと描画後の表示データを格納している配列のなすメモリ空間を2分し（一方をBuffer0、他方をBuffer1とする）、フレーム表示毎にメモリ配列を、

(Buffer0, Buffer1) = (描画ソース・データ, 描画後の表示用データ)

(Buffer0, Buffer1) = (描画後の表示用データ, 描画ソース・データ)

となるように、交互に切り替えているとする。従って、描画ソース・データを描画後の表示用データで上書きする事はないとする。更に、モデル簡略化の為、描画ソース・データの転送は省略する。

【0059】

また、モデル簡略化の為、テクスチャ・データは省略しており、描画コマンドも最高で2つしか格納しないものとし、コマンド0, 1の実行順番がどうであっても描画結果には影響しないものとする。

【0060】

(3) Graphics Rendering Unit

グラフィック描画ユニットに関するrun() methodの実装例を説明する。その実装例は図28乃至図32に例示される。Unified Memoryのコマンドフラグ0, 1をある一定周期毎にsync_burst_readにより読み出す(バス権獲得・解放動作)。何れか一方の値が1であった場合、内部状態変数render_startをtrueとし、バス権獲得を試み、獲得に成功したなら、描画コマンド、描画データをsync_burst_readで読み込みながら、描画処理を実行し、描画結果を内部バッファに書き込む(読み込み終了次第、コマンドフラグを0にクリアし、バス権解放する)。但し、モデル簡略化の為、例えコマンドフラグが両方とも0であっても、コマンドフラグ0に対応する描画コマンドのみを実行するものとし、描画ソースデータの転送は省略する。再び、バス権獲得を試み、獲得に成功したなら、描画結果をsync_burst_writeでUnified Memoryへ転送し、バス権を解放し、内部状態変数render_startをfalseとする。尚、内部状態変数render_startの初期値はfalseである。

【0061】

(4) Display Unit

表示ユニットに関するrun() methodの実装例を説明する。その実装例は図33及び図34に例示される。Unified Memoryから描画後の表示用データを読み込みCRTへ出力する。但し、モデル簡略化の為、水平同期のみを扱うものとする。これは、水平帰線期間内に描画が開始・終了する為のサイクル制約をパラメトリ

ック・モデルチェックで求める事を目的としているからである。

【0062】

さて、Display Unitは、内部変数display_startをtureに設定し、Unified Memoryから描画後の表示用データをsync_burst_readにて読み込む。データ読み込み終了後に内部変数display_startをfalseとして、ライン表示を実行し、ライン表示終了後から水平帰線期間に相当する適当なサイクルを消費し、再び、ライン表示を同じ手順で行うものとする。また、転送サイクルのみをモデル化すれば良いので、メモリアドレスは簡略的に表現し、常に同一メモリ空間から読み出しているものとする。

【0063】

《パラメトリック解析》

ここでパラメトリック解析の目標について説明する。本来、描画は1フレーム表示の表示期間及び、垂直帰線期間の間に1フレーム分の実行を終了しておれば良いと考えられるので、水平帰線期間での描画が満たすべき性質を勘案すると、検証内容としては下記条件を満たすサイクル制約の導出で十分である。即ち、「表示用データ取得終了後に描画を開始し、次の表示開始までに描画を終了する事がある。但し、描画開始・終了にデータ転送サイクルは含まれ、表示はnサイクル以内に終了する。」である。上記制約をRPCTL (Real Time Parametric Computation Tree Logic) で記述すると下記

```
EF {<display_end> (AF (<render_begin> true)
&& {AF (<render_end> true) AU (<display_begin> true)}}}
&& AG {[display_begin] (AF <= n ([display_end] true))}
```

の3個の制約のアンド条件となる。尚、表示データ取得終了迄に未実行の描画コマンドがUnified Memory上に常に存在する事が仮定できるものとする。ここで、nは適当な正整数であり、各変数は下記の通り、

display_begin: 内部変数display_startの0から1への立ち上がりイベント

display_end: 内部変数display_startの1から0への立ち下がりイベント

render_begin: 内部変数render_startの0から1への立ち上がりイベント

render_end: 内部変数render_startの1から0への立ち下がりイベント

を表す。

【0064】

パラメトリック・モデルチェッキングを高速化する目的で、R P C T L を

$$AG\{[display_begin](AF\leq n([display_end]true))\} \quad (*)$$

と、それ以外の部分に分離し、例えば、

$$EF\{<display_end>(EF(<render_begin>true)\&\& \{EF(<render_end>true)AU(<display_begin>true)\})\}$$

から実行し、パラメータ条件を導出し、導出されたパラメータから (*) の n の上限を求め、既に求めたパラメータ条件に矛盾しないパラメータ条件が導出できるまで、 n の値を減じて繰り返し、(*) のパラメトリック解析を実施するものとする。

【0065】

R P C T L やそれを用いたパラメトリック・モデルチェッキングの詳細は次の文献、“Akio Nakata and Teruo Higashino: ”Deriving Parameter Conditions for Periodic Timed Automata Satisfying Real-Time Temporal Logic Formulas”, Proc. of IFIP TC6/WG6.1 Int’l Conf. on Formal Techniques for Networked and Distributed Systems(FORTE2001), Cheju Island, Korea, Kluwer Academic Publishers, pp.151-166, Aug. 2001.”に記載ある。

【0066】

《中間表現への変換》

図35には中間表現へ変換工程の詳細が全体的に例示される。バスシステムをモデル化して記述されたジャバ言語記述はC (concurrent)－C F Gに変換され (Java2C-CFG)、C－C F GはC－T N F Aに変換され (C-CFG2C-TNFA)、C－T N F AはT N F Aに変換される (C-TNFA2TNFA)。パラメトリック・モデルチェッキングを経てC－C F GがH D Lに変換される (C-CFG2HDL)。尚、上記Java2C-CFG等の標記において、数字“2”は“to”を意味するものと理解されたい。

【0067】

《Java2C-CFG》

ジャバ言語記述からC－C F Gへの変換アルゴリズムの概要を説明する。各ru

n() メソッドに対応するCFGを生成する。特に、クロック同期メソッド (consume_clock) はクロック境界として識別し、呼び出しメソッドは呼び出し関係を表すノードを用いて表現する。各run()メソッドに対応するCFGが生成されると、forkノードを設け、そのノードから各CFGの開始ノードへのfork枝を付加する。特に、バス上の各デバイスにrun()メソッドが対応するものとしてJavaが記述されている事を前提とし、メモリデバイスに関しては、CFG生成対象外である事の指定を与えるものとする。

【0068】

次いで、呼び出しメソッドのCFGを作成し、呼び出しメソッドがsynchronizedブロック内に存在する場合は、呼び出しメソッド内のsynchronizedをCFGから削除する。特に、呼び出しメソッドがRegisterクラス内の通信メソッドである場合はCFGの作成を行わず、インスタンス関係からどのデバイス内のどのレジスタへのアクセスかと通信メソッドの名前だけを識別し、その情報をCFGに保持する。但し、図ではどのレジスタへのアクセスかの情報は省略している。尚、通信メソッドが内部にクロック同期メソッドを含む場合は、その出力枝にクロック境界をマークする。更に、呼び出しメソッド全体のサイクル制約を導出する以外は、呼び出しメソッドをインライニングする。この例では、Renderingメソッド、Displayメソッドはインライニングしない。

【0069】

synchronizedを予め定めたCFGに展開し、ハード合成用の固定優先度スケジューラをFSM (Finite State Machine) の形式でスケルトンとして予め与えてあるライブラリと実際にCFGを生成したrun()メソッドの個数を用いて、自動生成する。ここで、CFGはハード合成用とパラメトリック・モデルチェッキング用の2種作成する。

【0070】

尚、固定値伝播や、代入によるローカル変数消去等のコードレベル最適化をCFG上で実施する。特に、信号の入力・出力・入出力はメンバ変数で表し、端子方向に関しては、プログラム内での代入文で右辺にあるのか左辺にあるのかや条件文での変数の用いられ方を解析して決定する。入出力信号として識別された場

合、データ依存性を考慮して、入力と出力に適当な信号のリネームを行って分離する。

【0071】

Java 2 C-CFGに関し、以下で、Java記述例に沿って、各run()メソッドのCFG生成結果、C-CFG生成結果、及び固定優先度・スケジューラのFSMについて説明する。

【0072】

先ずCFGの形式について説明する。図36にはC-CFGの形式が例示される。各CFGは分岐枝の開始と終了を表すノードと、ループ枝の開始と終了を表すノードを含み、分岐条件、ループ終了条件を対応する枝に付加した形式とする。特に、クロック境界ノードを枝上に付加する事でクロック境界を表現する。また、メソッドコールに対応するノードを用いる事で、サブCFGとのリンクを表現する。各CFGの並列実行を表すforkノードはCFGへの頂点へ付加する事で表現する。

【0073】

synchronizedの扱い（ハード合成用）について説明する。図37に例示されるように、synchronizedの開始をCFG上でbegin_syncとラベル付けされたノードとして表し、synchronizedの終了をend_syncとラベル付けされたノードでし、CFGを生成。その後、その両ノードを夫々下図に示すように変換する。

【0074】

synchronizedの扱い（パラメトリック・モデルチェック用）について説明する。図38に例示されるように、synchronizedの開始をCFG上でBegin_syncとラベル付けされたノードとして表し、synchronizedの終了をEnd_syncとラベル付けされたノードでし、CFGを生成。その後、その両ノードを夫々図38に示すように変換する。

【0075】

上記に従うと、前記Command Interfaceの場合、CFGは図39に示される。getWriteSignal()メソッドは、シミュレーションを実施する為に行った記述であり、処理系への入力では外部からの入力信号write_reqであるとして修正されて

いるものとし、入力テストベクタの個数を表すdcom_indexを補正する記述も削除されているものとする。また、drawing_commandsも配列として表現されているが、これもシミュレーションを実施する為になされたものであり、内部変数input_commandに内部変数drawing_commandsが入力されている記述が処理系に入力されているものとして扱う。

【0076】

前記Command Interfaceに関し、ハード合成用のCFGは図40に示され、パラメトリック・モデルチェック用のCFGは図41に例示される。

【0077】

前記グラフィック描画ユニット (Graphics Rendering Unit) に関し、図42及び図43にはそのCFGが例示され、図44及び図45にはそのパラメトリック・モデルチェック用のCFGが例示される。

【0078】

前記表示ユニット (Display Unit) に関し、図46にはそのCFGが例示され、図47にはそのパラメトリック・モデルチェック用のCFGが例示される。

【0079】

図48にはコマンドインタフェース (Command Interface) のCFG、グラフィック描画ユニット (Graphics Rendering Unit) のCFG、及び前記表示ユニット (Display Unit) のCFGにおける夫々の開始ノードの結合状態が例示される。

【0080】

固定優先度スケジューラ (Fixed Priority Scheduler) について説明する。実際にトップレベルのCFGを生成したrunメソッドの数を識別し、夫々のデバイスにバス権取得通知を行うための信号locked_iの組みからなる状態を作成する。単一バスシステムであるため、それら状態は、1つのlocked_iのみが1でその他が0であるものを構成すれば良い。また、全てのlocked_iが0である状態も作成する。優先度情報に基づき、バス権要求信号lock_iを受けて行う状態遷移枝を設ければバス権管理用の固定優先度スケジューラが構成できる。特に

、全てのlocked_iが0である状態をリセット解除時の開始状態とする。
 また、各状態遷移は $l \leq t \leq kl$ の時間制約が付加されているものとする。このパラメータは後のパラメトリック・モデルチェッキングで決定されるパラメータである。尚、例えば、 $kl=2$ とし、各遷移を2クロック遷移とした場合は、出力値は変化が起こるまで、前の値を保持しているものとして解釈するものとする。

【0081】

図49には本例に対応する固定優先度スケジューラが例示される。同図における信号の意味は、

lock_1: Command Interfaceからのバス権要求信号

lock_2: Graphics Rendering Unitからのバス権要求信号

lock_3: Display Unitからのバス権要求信号

locked_1: Command Interfaceへのバス権取得通知信号

locked_2: Graphics Rendering Unitへのバス権取得通知信号

locked_3: Display Unitへのバス権取得通知信号

である。尚、バス権取得の優先度が、

Display Unit > Command Interface > Graphics Rendering Unit

であるので、遷移条件は、

lock_3: Display Unitへの遷移

!lock_3 && lock_1: Command Interfaceへの遷移条件

!lock_3 && !lock_1 && lock_2: Graphics Rendering Unitへの遷移条件

otherwise: 全てのlocked_iが0である状態への遷移条件

となる。

【0082】

《Abstraction of each CFG》

夫々のCFGの抽象化処理について説明する。先ず、そのアルゴリズムの概要を説明する。各BasicBlockに対して、 $0 \leq t \leq ki$ なる時間遷移条件を付加し、Begin_syncノードとEnd_syncノードに対して $0 \leq t \leq kl$ を付加し、通信メソッドを表すノードに対して、バースト開始に $l \leq t \leq kb1$ 、バースト動作に $0 \leq t \leq kb2$ 、バースト終了に $0 \leq t \leq kb3$ を付加する。但し、通信メソッドを表すノードに対しては、

バースト開始が2サイクル以上、バースト動作時は1サイクル以上、バースト終了が1サイクル以上を要すると考え、後述のノードのマージによる抽象化を行う際には、サイクル制約の補正を行う。Begin_syncとEnd_syncノード以外の制約を付加したノードをクロック境界と見做して抽象化を行う。

【0083】

通信メソッドは動作内容を全て抽象化し、連続する場合は、1つのノードにまとめ、例えばバースト開始($1 \leq t \leq kb1$)、クロック境界($t=1$)、(バースト動作($0 \leq t \leq kb2$) + クロック境界($t=1$)) 3回、バースト終了($0 \leq t \leq kb3$)、クロック境界($t=1$)とすると、

$$6 \leq t \leq (kb1+1-1)+3(kb2+1-1)+(kb3+1-1) = kb1+3kb2+kb3 \quad (>=2+3*1+1 = 6)$$

なるサイクル制約を持つクロック境界として1つにまとめる。また、分岐ノードは全て非決定分岐であるとする。Basic Blockノードに関しては、入力RPCTLで用いる変数のみ残し他は抽象化する。分岐ノード下位に存在するBasic Blockに対応する部分が異なる分岐で同じ場合は、1つを残して他を削除する。

【0084】

特にループノードに関しては、break、continueを含まない固定回数ループの場合はアンローリングを実施し、それ以外の場合は、ループを抜ける条件枝の直前に全体で $0 \leq t \leq kloop$ のサイクル制約を持つようなCFGノードの集合を設ける事でループを削除する。ここで、kloopの満たすべき条件を別に算出するものとする。更に連続するクロック境界はパーコレーション・ベースの移動を行う事で、1つに纏め、遷移サイクル制約を更新する。

【0085】

ここで、各クロック境界のパラメータ化により、クロック境界に与えた遷移サイクル制約の意味は、クロック境界から次のクロック境界への遷移に対するサイクル制約である。

【0086】

コマンドインタフェース (Command Interface) のCFGに対する抽象化の様子が順を追って図50乃至図56に例示される。図57にはグラフィック描画ユニット (Graphics Rendering Unit) のCFGに対する抽象化処理の結果が例示

される。図58には表示ユニット (Display Unit) のCFGに対する抽象化処理の結果が例示される。

【0087】

《C-CFG2C-TNFA》

C-TNFAからTNFAへの変換処理について説明する。TNFAとは時間オートマトンであり、各遷移が、

$s \xrightarrow{a@t[P(t)]} s'$

の形式で表されるものである。ここで、各記号の意味は、

s, s' : 状態、

a : 動作(省略可能)、

$@t$: 状態 s を訪れてから動作 a を実行するまでの経過時間を t に代入、

$P(t)$: 時間制約(t に関する線形不等式の論理結合)、

である。また、その意味は「 s から t 単位時間経過後に状態 s' に遷移。ただし、 t は時間制約 $P(t)$ を満たす場合に限る」である。

【0088】

C-TNFAは、複数のTNFAが並列動作するモデルであり、動作syncは高々1つのTNFAしか実行できないモデルである。特に連続する動作syncに対しては他のTNFAが割り込めない。尚、各TNFAに対して、初期状態から初期状態への遷移に対して、上限を与える為の時間制約を課す事が可能である。

【0089】

次にC-CFGからC-TNFAへの変換処理について説明する。先ず、その変換アルゴリズムの概要を説明する。Begin_sync、End_syncで挟まれた部分を識別し、syncブロックとする。Syncブロック内に存在しないクロック境界ノードにステート割り当て候補とし、またsyncブロック内であっても検証性質に必要な信号がBasic Blockとして与えられている場合はその前後のクロック境界ノードにステートを割り当て候補とし、最後にBegin_sync、End_syncの変換で導入したクロック境界をステート割り当て候補とする。得られたステート割り当て候補を重複が無いようにし、ステート割り当てを行い、各ステートからステートまでをDFS (Depth First Search) でトラバースする事でTNFAへの変換を行う

。得られたTNFAに対して、syncブロックに対応する遷移を検出し、sync遷移とする。連続するsync遷移でない遷移は1つの遷移に纏める事で状態数の削減を行う。

【0090】

コマンドインタフェース (Command Interface) のCFGに対するTNFAへの変換の様子が順を追って図59乃至図61に例示される。

【0091】

図62乃至図65にはグラフィック描画ユニット (Graphics Rendering Unit) のCFGに対するTNFAへの変換の様子が順を追って例示される。図65において、検証性質は、「表示用データ取得終了後に描画を開始し、次の表示開始までに描画を終了する事がある。但し、描画開始・終了にデータ転送サイクルは含まれ、表示はnサイクル以内に終了する。」であり、これは、描画が開始される事がもしあればという前提にたった検証性質であるから、Graphics Rendering Unitに対応するTNFAにおいて、表示データ取得終了迄に未実行の描画コマンドがUnified Memory上に存在しない事を表すR2からR1への遷移はこの検証には不要となる。従って、これは削除されている。

【0092】

図66乃至図68には表示ユニット (Display Unit) のCFGに対するTNFAへの変換の様子が順を追って例示される。

【0093】

《C-TNFA2TNFA》

C-TNFAからTNFAへの変換アルゴリズムの概要を説明する。先ず、下記方針(1)～(5)に従ってC-TNFAから積TNFAを構成する。

【0094】

(1) sync動作とそれ以外の動作は互いにインタリーブされる。ここで、sync動作を行っている間にバス権をとる必要が無い動作、即ちsyncでない動作を(一般に複数)並行して実行できることを表現。但し、sync動作を1遷移実行する間の他のTNFAの並行遷移に遷移枝を通る回数に制限を設け、それを満たす各遷移の遷移時間の上限を導出する。特に、回数は1から始め、意味のある解が得られ

るまで、回数を1ずつインクリメントするものとする。

【0095】

(2) 複数のsync動作が実行可能な場合は静的優先度が最も高いものによって遷移する。ここで、動いていない状態 s に関しては $\text{age}(s, t)$ (状態 s で時間 t だけが経過した状態) に遷移。ただし、連続するsync遷移の間には他の動作は割り込めない。

【0096】

(3) $\text{age}(s, t)$ 以降の遷移に関しては、時間 t に関して下記補正を行う。即ち、 $s - [P(t_1)] \rightarrow s'$ ならば $\text{age}(s, t) - [P(t+t_1)] \rightarrow s' [t+t_1/t_1]$ 、とする。ここで、 $s[e/t]$: 状態 s からの遷移条件に現れる変数 t を式 e で置き換えることを表す。尚、再構成した $P(t)$ が矛盾した式になっている場合、その遷移は削除し、その遷移へのみ到達する状態全てを削除するものとする。

【0097】

(4) 各遷移の時間変数 t は下記に示すように、異なる名前に変更する。即ち、 $s - [P(t)] \rightarrow s' - [Q(t)] \rightarrow s''$ は $s - [P(t_1)] \rightarrow s' - [Q(t_2)] \rightarrow s''$ に変更する。 $\text{age}(s, t)$ に関する補正によって、各遷移条件はそれ以前の遷移の時間変数も含む式になるため、それらを区別するために、この変更が必要となる。

【0098】

(5) 上限を与えたTNFAがある場合、構成後のTNFAでその初期状態を含む状態間の遷移がその上限を満たさない場合、その途中にある状態で上限を満たす遷移内に存在する状態を削除対象外とし、削除対象外とならなかった状態を削除する。

【0099】

尚、Abstraction of each CFGのステップで抽象化対象から除外されたRPC TLに用いられている変数を伴う遷移が現れるとその次の状態まで構成し、上記構成方針での作成での切りの良いところで、Abstraction of TNFAをコールし、構成中のTNFAの抽象化を行う。

【0100】

次にC-TNFAからTNFAへの変換処理を詳細に説明する。まず、遷移回

数の仮定による遷移時間の上限を決定する。即ち、各TNFAの強連結成分を求め、強連結成分内での遷移枝を1回のみ通る各遷移の下限遷移時間の総計が最大となる最長パスと、そのパス上の頂点でその後段の頂点への遷移時間の下限が最小の頂点を求め、最長パスにその後段頂点へのパスを追加したパスの2つを求める。次いで、夫々のパスの下限遷移時間の総計を求める。これにより、

Command Interface: 22 23

Graphics Rendering Unit: 28 29

Display Unit: 11 12

が求まる。さて、各TNFAの2つ目の下限遷移時間の総計から1引いた値、

Command Interface: 22

Graphics Rendering Unit: 28

Display Unit: 11

を求める。各TNFAの各遷移の上限時間を、他のTNFAに対して求めた値の最小値を与える事で決定し、得られた時間遷移の上限値が矛盾したものとなっていないかを、既に上限が与えられている遷移の上限値と比較する事で調べ、もし、矛盾がでたなら2回に増加させる。そうでなければ、上限が与えられていない遷移に求めた上限を与える事で線形不等式の論理結合を構成する。特に、回数を増加させる場合、最長パスの下限遷移時間の総計と回数の積を取った値と、その値に先に求めた最小の遷移時間を加えた値を算出すればよい。

【0101】

尚、パラメトリック・モデルチェッキングを実施し、パラメータ条件が非負整数解を持たない場合にも回数の増加を行って処理をすすめる事とする。ここで、非負整数解の算出には、線形計画法を用いる。具体的には、フリーソフトLP-SOLVを用いて得られたパラメータ条件を満たす範囲で、パラメータの和が最小となる解を算出する。仮に、得られた解が意味をなさない場合は、パラメータの最小値を付加するなどして対処する。

【0102】

さて、本例の場合、遷移枝を一回のみ遷移可能とした仮定では、各TNFAの遷移時間の上限として

Command Interface : 11

Graphics Rendering Unit : 11

Display Unit : 22

が求まる。また、矛盾が起こることなく、以下に示す線形不等式の論理結合、

$$0 \leq k_1 + k_2 + k_{loop1} \leq 7 \ \&\& \ 5 \leq kb_1 + kb_2 + kb_3 + k_1 \leq 8 \ \&\& \ 7 \leq kb_1 + 3kb_2 + kb_3 + k_1 \leq 11 \ \&\& \ 1 \leq k_1 \leq 11 \ \&\& \ 1 \leq k_1 + k_2 + k_3 + k_{loop1} + k_1 \leq 6 \ \&\& \ 3 \leq k_4 \leq 11 \ \&\& \ 5 \leq kb_1 + kb_2 + kb_3 + k_1 \leq 11 \ \&\& \ 6 \leq kb_1 + 4kb_2 + kb_3 + 3kr_1 \leq 12 \ \&\& \ 1 \leq 3kr_2 + k_1 \leq 8 \ \&\& \ 9 \leq kb_1 + 5kb_2 + kb_3 + k_1 \leq 12 \ \&\& \ kr_1 = kb_2 - 1 \ \&\& \ 8 \leq kb_1 + 5kb_2 + kb_3 \leq 24 \ \&\& \ 1 \leq k_1 + 6kd \leq 19$$

が得られる。また、遷移枝を 2 回まで遷移可能とした場合、各 TNFA の遷移時間の上限として

Command Interface : 22

Graphics Rendering Unit : 22

Display Unit : 44

が求まり、以下に示す線形不等式の論理結合、

$$0 \leq k_1 + k_2 + k_{loop1} \leq 18 \ \&\& \ 5 \leq kb_1 + kb_2 + kb_3 + k_1 \leq 19 \ \&\& \ 7 \leq kb_1 + 3kb_2 + kb_3 + k_1 \leq 22 \ \&\& \ 1 \leq k_1 \leq 22 \ \&\& \ 1 \leq k_1 + k_2 + k_3 + k_{loop1} + k_1 \leq 17 \ \&\& \ 3 \leq k_4 \leq 2 \ \&\& \ 5 \leq kb_1 + kb_2 + kb_3 + k_1 \leq 22 \ \&\& \ 6 \leq kb_1 + 4kb_2 + kb_3 + 3kr_1 \leq 23 \ \&\& \ 1 \leq 3kr_2 + k_1 \leq 19 \ \&\& \ 9 \leq kb_1 + 5kb_2 + kb_3 + k_1 \leq 23 \ \&\& \ kr_1 = kb_2 - 1 \ \&\& \ 8 \leq kb_1 + 5kb_2 + kb_3 \leq 46 \ \&\& \ 1 \leq k_1 + 6kd \leq 41$$

が得られる。

【0103】

この工程は、Assume Guarantee Reasoning と呼ばれる手法であり、パラメトリック解析での実施は公知ではない。

【0104】

図 69 には図 61、図 64、図 68 で求めた TNFA の積 TNFA の構成例が示される。これに基づいて TNFA を求める過程が図 70 乃至図 76 に示される。

【0105】

《Abstraction of TNFA》

TNFA の抽象化処理について説明する。まず、そのアルゴリズムの概要を説

明する。TNFAで抽象化時に残した変数への代入を行っている遷移枝と、その遷移枝の始点ノード及び終点ノードのみを残し、残すべき頂点を始点として、次に残すべき頂点が現れるまで頂点と辺をトラバースしながら遷移時間を再計算する事で、その他の頂点を全て抽象化する。但し、葉に対応する状態は抽象化の対象から除外する。

【0106】

先の例では抽象化の実施がまだ起こらない段階までしかTNFAを構成しなかったもので、抽象化が必要となる様先の例を意図的に若干修正して、図77乃至図79にC-TNFA2TNFAでAbstraction of TNFAを適時コールした場合の処理の経過の一部を示す。ここでは、klには具体的な値は与えないものとする。

【0107】

《パラメトリック解析結果》

前記文献に述べられているアルゴリズムを用いて、遷移を2回以上通らない制約で、探索深さ最大値16として、以下の検証性質

```
EF (<displayend>( (AF(<renderbegin>true))
and ((AF(<renderend>true)) AU (<displaybegin>true))))
```

に関してパラメタ条件を導出すると、以下の条件、

$$0 \leq kd \ \&\& \ 0 \leq kr2 \ \&\& \ 0 \leq kr1 \ \&\& \ 0 \leq kb3 \ \&\& \ 0 \leq kb2 \ \&\& \ 0 \leq kb1 \ \&\& \ 0 \leq kloop1 \ \&\& \ 0 \leq k5 \ \&\& \ 4 \leq k4 \ \&\& \ 0 \leq k3 \ \&\& \ 0 \leq k1 \ \&\& \ 0 \leq k2 \ \&\& \ 12 \leq kb1+9kb2+kb3 \ \&\& \ 4 \leq k1)$$

を得る。これと先に求めた、遷移を2回以上通らないという制約から得られたパラメタ条件

$$0 \leq k1+k2+kloop1 \leq 7 \ \&\& \ 5 \leq kb1+kb2+kb3+k1 \leq 8 \ \&\& \ 7 \leq kb1+3kb2+kb3+k1 \leq 11 \ \&\& \ 1 \leq k1 \leq 11 \ \&\& \ 1 \leq k1+k2+k3+kloop1+k1 \leq 6 \ \&\& \ 3 \leq k4 \leq 11 \ \&\& \ 5 \leq kb1+kb2+kb3+k1 \leq 11 \ \&\& \ 6 \leq kb1+4kb2+kb3+3kr1 \leq 12 \ \&\& \ 1 \leq 3kr2+k1 \leq 8 \ \&\& \ 9 \leq kb1+5kb2+kb3+k1 \leq 12 \ \&\& \ kr1=kb2-1 \ \&\& \ 8 \leq kb1+5kb2+kb3 \leq 24 \ \&\& \ 1 \leq k1+6kd \leq 19$$

との論理積を取って、得られたパラメタ条件に対して、目的関数を各パラメタ値の合計として、これを最小化するという線形計画問題をフリーソフトLP_SOLVEにて解かせると、図80の解を得る。

【0108】

図80の結果では、バスアクセス終了通知が0サイクルで行われなければならない、また、描画サイクルkr2と表示サイクルkdが0であるので、意味のある解とは言えないので、kb1=2, kb2=1, kb3=1の制約を追加し、図81の解を得た。

【0109】

図81の結果でも、描画サイクルkr2と表示サイクルkdが0であるので、意味のある解とは言えないので、kb1=2, kb2=1, kb3=1の制約に加えてさらに、kr1以外のパラメタ変数は1以上という制約を追加し、図82の解を得た。以後、このパラメタ値を採用して議論を進める。

【0110】

《ハード合成》

ハードウェア合成処理についてその概要を先ず説明する。(1)～(11)の方針にてハードウェアの生成が行なわれる。

【0111】

(1) 事前にRegisterクラスに登録された通信メソッドの種類からバスコマンドの割り当てを行う。

【0112】

(2) ユーザ情報として、どのデバイスにどの共有レジスタが存在するかを受け付け、各デバイス内のレジスタ数に対応するデバイス内アドレスを割り当てる。

【0113】

(3) 共有レジスタが割付られたデバイス (run()メソッド) の数を取得し、共有レジスタが割り当てられたデバイスにグローバルなアドレスを割り当てる。

【0114】

(4) インライニング (展開) を実施しなかった通信メソッド以外のメソッドのインライニングを行う。(通信メソッドと指定したメソッド以外のメソッドは事前にインライニングされている事に注意。)

【0115】

(5) パラメトリック解析結果から得られた、Basic Blockの実行サイクルを各デバイスの合成用CFGに反映する。

【0116】

(6) 各デバイス夫々のCFGを変形し、通信メソッドとそれ以外のコントロールフロー部が互いに通信するモデルに変換する。

【0117】

(7) 前記(6)で得られた変換後のCFG内の通信メソッドに対応する部分に、バスコマンド生成記述を挿入する。通信メソッドは、グローバルアドレス、共有レジスタアドレス、バスコマンドを出力し、データ読み出し・書き込みの何れかを実行する記述からなるCFGとなり、それらの信号生成・受理に要するサイクルは、パラメトリック解析の結果により決定される。

【0118】

(8) 共有レジスタアドレスから、各デバイス内のレジスタの配列インデックスを生成するアドレスデコーダとグローバル・アドレスを識別するアドレスデコーダを共有レジスタが割り当てられたデバイス内に生成し、出力トライステートを共有レジスタの出力に接続し、トライステート・イネーブル信号として、各デバイスからのデータ入力ステージを表す信号の論理和を用いる。共有レジスタの入力に関しては、取り込み動作が起こらない限り内部変数は前の値を保持する為バスから信号を接続すれば良よく、各デバイスからのデータ入力ステージを表す信号の論理和を元にデータ取り込み判定を行う。但し、これらはCFGとして表現する。(自デバイス内に共有レジスタがあったとしても、この方式に従って、アクセスが行われているものとする。)

【0119】

(9) 変形等で得られた各CFGの信号通信関係(入力、出力)を識別し(バスは入出力)、夫々をモジュールとして識別する。

【0120】

(10) 各CFGを「サイクルアキュレートなプログラム記述からのハード合成」に従って、HDLへと変換する。

【0121】

(11) バスに対してリピータ回路をHDLの形式で挿入する。

【0122】

前記バスコマンドの割り当てに関し、本例で登録したバスアクセス・メソッドは、sync_read、sync_write、sync_burst_read、sync_burst_write、endBurstAccessであるが、各run()メソッドのCFGを解析すると、実際に用いられているバスアクセスメソッドは、sync_burst_read、sync_burst_write、endBurstAccesssであるため、バスコマンドを

```
sync_burst_read  2' b00
sync_burst_write 2' b01
endBurstAccess  2' b10
NOP 2' b11
```

のように割り当てる。

【0 1 2 3】

上記アドレス割り当てに関しては、共有レジスタはUnified Memoryにのみ割り付けられ、Unified Memoryの仕様から割り付けられた共有レジスタは

mem_con_reg.current_value[0]：コマンドフラグ0、
mem_con_reg.current_value[1]：描画コマンド0、
mem_con_reg.current_value[2]：コマンドフラグ1、
mem_con_reg.current_value[3]：描画コマンド1、
mem_con_reg.current_value[4]：描画ソース・データ、及び描画後の表示用データ、
mem_con_reg.current_value[5]：描画ソース・データ、及び描画後の表示用データ、
mem_con_reg.current_value[6]：描画ソース・データ、及び描画後の表示用データ、
mem_con_reg.current_value[7]：描画ソース・データ、及び描画後の表示用データ、
mem_con_reg.current_value[8]：描画ソース・データ、及び描画後の表示用データ、
mem_con_reg.current_value[9]：描画ソース・データ、及び描画後の表示用データ、となる。尚、ユーザ情報から各レジスタのビット幅が指定されているものと

し、それぞれ 3 2 ビット (unsigned int) であるとする。

【0 1 2 4】

バスアクセス・メソッドはレジスタ内のバイトアクセス等を行わないので、各レジスタのアドレスを

mem_con_reg.current_value[0] : 4' b0000、
mem_con_reg.current_value[1] : 4' b0001、
mem_con_reg.current_value[2] : 4' b0010、
mem_con_reg.current_value[3] : 4' b0011、
mem_con_reg.current_value[4] : 4' b0100、
mem_con_reg.current_value[5] : 4' b0101、
mem_con_reg.current_value[6] : 4' b0110、
mem_con_reg.current_value[7] : 4' b0111、
mem_con_reg.current_value[8] : 4' b1000、
mem_con_reg.current_value[9] : 4' b1001、のように決定する。

【0 1 2 5】

また、アドレス割り当てに関し、共有レジスタが割り当てられたデバイスは Unified Memory しか存在しないため、グローバルアドレスの割付を行わない。

【0 1 2 6】

尚、本例では 1 つのデバイスの中に共有レジスタの割り当てを行っているが、以降の説明に於いてこれは一般性を欠くものではない。それは、以降に処理手順を示す事で明らかとなる。

【0 1 2 7】

前記 BasicBlock への実行サイクルの割り当てに関しては、ここでは、Command Interface のみ取り上げて説明を行う。特に、先に求めたパラメータ値から、 $k1 = 4$, $kb1 = 2$, $kb2 = 1$, $kb3 = 1$, $k1 = k2 = k3 = 1$ 、として説明を行う。パラメータ条件から、各 BasicBlock の下に割り当てられたサイクル数のクロック境界を挿入する。ここでは、バスアクセス・メソッドを扱う対象としていない事に注意。更に、与えたパラメータで、例を示しているが一般性を失わない事に注意。次ページに変形後の C F G を示す。尚、 $k1$ の値は、直接各デバイスのハード合成

用CFGには影響を与えない事にも注意。これは、固定優先度スケジューラにのみ影響する。図83にはBasicBlockへの実行サイクルの割り当ての例が示される。

【0128】

固定優先度スケジューラの修正に関しては、既に得られているFSM (Finite State Machine) の遷移サイクルをklに変更する、即ち、クロック境界を各遷移枝に4つ付加すれば良い。ここからのHDL生成は、各状態からの遷移を分岐ノードに表現し直し、かつ各状態での信号代入はその状態への遷移枝上での最後のクロック境界直下にBasicBlockを設け、そこにレジスタ代入文を記述する形式で置き換える事で、このFMS自体をCFGと見做す事で、「サイクルアキュレートな記述からのHDL生成」を用いる事で可能である。図84に示す固定優先度スケジューラの一部を用いて変形後のCFGが図85に示される。

【0129】

CFGの変形に関しては、元のCFGからバスアクセス・メソッドの括りだしを行い、バスアクセス・メソッドと元のCFGとの通信ノードを設る。具体的には、下記を行う。

【0130】

1) ハード合成用のCFG生成段階で、クロック境界を含むバスアクセス・メソッドのクロックをバスアクセス・メソッドを表すノードの直下にクロック境界を追加したが、これを削除する。

【0131】

2) バスアクセス・メソッドを表すノードを下記処理を行うBasic Blockに置き換える。

① 出力信号 start_comm に1を代入する。

② バスコマンドを出力信号bus_cmdに代入する。バースト転送の場合、初期サイクルでは、

<1> 出力信号initの0を代入

<2> アクセスするデバイスのアドレスと共有レジスタのアドレスを連結して出力信号addressに代入

<3> クロック境界挿入×n b f

<4> リードメソッドなら、代入すべき変数に入力信号data_inを代入

<5> ライトメソッドなら、出力すべき値または変数を出力信号data_outに代入

<6> クロック境界×m b f、を行う。ここで、 $n b f + m b f = k b 1$ である。

この例では、 $n b f = m b f = 1$ とする。

バースト転送の場合、転送終了では、

<1> クロック境界挿入×n、を行う。ここで、 $n = k b 3$ である。ここでは、 $n = 1$ とする。

バースト転送の場合、それ以外では、

<1> 出力信号initに1を代入

<2> アクセスするデバイスのアドレスと共有レジスタのアドレスを連結して出力信号addressに代入

<3> クロック境界挿入×n b

<4> リードメソッドなら、代入すべき変数に入力信号data_inを代入

<5> ライトメソッドなら、出力すべき値または変数を出力信号data_outに代入

<6> クロック境界×m bを行う。ここで、 $n b + m b = k b 2$ である。この例では、 $n b = 0$ 、 $m b = 1$ とする。

シングル転送の場合、

<1> アクセスするデバイスのアドレスと共有レジスタのアドレスを連結して出力信号addressに代入

<2> クロック境界挿入×n s

<3> リードメソッドなら、代入すべき変数に入力信号data_inを代入

<4> ライトメソッドなら、出力すべき値または変数を出力信号data_outに代入

<5> クロック境界挿入×m s、を行う。ここで、 $n s + m s = k s$ である。

③ 出力信号 start_comm に0を代入する。

【0132】

3) バスアクセスメソッドを表す孤立ノードを作成し、置き換えて生成したBasic Blockとの通信ノードを設ける。通信ノードとして、

start_comm: Basic Block → 孤立ノード

bus_cmd: Basic Block → 孤立ノード

address: Basic Block → 孤立ノード

data_in: 孤立ノード → Basic Block

data_out: Basic Block → 孤立ノード、を設ける。

【0 1 3 3】

図 8 6 及び図 8 7 には変形後のCommand Interfaceの C F G の一部が示される。図 8 7 において、クロック境界の傍らに記載された変数はパラメータであり、その数だけクロック境界を生成することを意味する。

【0 1 3 4】

孤立ノードへの C F G の割り当てに関しては、図 8 8 の C F G を自動生成する。図 8 8 において、特に、AD_BUSはアドレスバスを、D_BUSはデータバスを表し、data_in_en_iはデータ入力ステージを、data_out_en_iはデータ出力ステージを表す。ここで、iは各デバイスのインデックスを表す。また、アドレスバスのバス幅は、割り当てたバスコマンドのビット幅とアドレスのビット幅の合計とし、データバスはアクセスするレジスタのビット幅とする。また、クロック境界の傍にかかれた変数はパラメータであり、その数だけクロック境界を生成する事を意味する。尚、各変数の値は、C F G の変形過程で用いた変数の値と等しい。

【0 1 3 5】

共有レジスタに関しては、図 8 9 の記述をインライン展開したものに対応する C F G の生成を行えばよい。但し、入出力変数であるAD_BUS、D_BUSの入力、出力への分離は行わず、また C F G 上での最適化に於いては、AD_BUS、D_BUSに対する変数最適化は実施しないものとする。図 8 9、図 9 0 及び図 9 1 には共有レジスタに関する擬似 C 記述である。サイクルアキュレートな記述からの H D L 生成に関しては本発明者による先の出願（特願 2 0 0 2 - 3 0 0 0 7 3）を参照することができる。

【0 1 3 6】

以上本発明者によってなされた発明を実施形態に基づいて具体的に説明したが、本発明はそれ限定されるものではなく、その要旨を逸脱しない範囲において種々変更可能であることは言うまでもない。

【0137】

例えば、積オートマトンの構成に関しては、TNFA積オートマトンを構成するとき、遷移枝の通過回数でAssume Guarantee Reasoningを実施したが、本来は、sync動作の通過回数で行うべきである。理由は、sync動作はバスアクセス動作を意味しており、並列に動いているTNFAが何回sync動作を実施したかは、そのTNFAに対応するデバイスがバスアクセスを何回行ったかに対応するからであり、Refinement及びパラメトリック・モデルチェッキングで得られるのは、遷移サイクルの条件のみならず、その検証性質を実施している間に各デバイスが高々どれだけバスアクセスを行うかの情報も得られるからである。但し、この手法であると、組み合わせ爆発を起し得るので、何らかの高速な枝切りが必要となる。

【0138】

積オートマトンの構成に関しては、今回は、パラメトリック・モデルチェッキングの停止性を吟味せず、バウンデッド・モデルチェッキングとして、検証性質を満たす必要条件を求める事を行ったが、本来は十分を求める必要がある。その為、アルゴリズムの停止性に関する研究を更に行う必要がある。

【0139】

バスモデルの拡張に関しては、単一双方向バスのみではなく、単方向バスや、ローカルバスを含むもの、またバス・ブリッジを介してバスが階層化されているような複数バスシステムを扱うようにしてもよい。

【0140】

また、水平帰線期間のみを扱ったが、本来は垂直帰線期間もモデル化して、1フレームを表示している間に最低1フレーム分の描画が終了するための十分条件を求める必要がある。

【0141】

【発明の効果】

本願において開示される発明のうち代表的なものによって得られる効果を簡単に説明すれば下記の通りである。

【0142】

すなわち、並列動作を記述可能なプログラム言語を用いてバス・システム等のハードウェア設計の工数を低減、更にはハードウェア設計から論理合成までの工数を低減することができる。

【0143】

並列動作を記述可能なプログラム言語とパラメトリック・モデルチェッキングを用いた、実時間制約を満たすバス・システムの新規設計手法を提供することができる。

【0144】

モデルチェッキング技術とハードウェア合成技術の融合による新たな設計手法を提供することができる。

【0145】

実時間制約を有するバス・システムに対する並列動作記述可能なプログラム言語によるモデル化及びパラメトリック・モデルチェッキングによる検証、更にはハードウェア合成を実現することができる。

【図面の簡単な説明】

【図1】

本発明に係る設計方法の全体を例示するフローチャートである。

【図2】

単一バス・システムのジャバ言語によるモデル化のデザインパターンを例示する説明図である。

【図3】

モデル化におけるクロック同期メカニズムとしてバリア同期によるクロック同期メソッドを用いた例を示す説明図である。

【図4】

レジスタへの値書込みを実現するモデル化の記述を例示する説明図である。

【図5】

バス権獲得を管理するメソッドを例示する説明図である。

【図6】

図5の各メソッドの呼び出し関係を表すコールグラフである。

【図 7】

バス権獲得メカニズムのジャバ言語記述を例示する説明図である。

【図 8】

図 7 の続きを示すジャバ言語記述の説明図である。

【図 9】

バス権解放を管理するメソッドの説明図である。

【図 10】

図 9 の各メソッドの呼び出し関係を表すコールグラフである。

【図 11】

バス権解放メカニズムのジャバ言語記述を例示する説明図である。

【図 12】

sync_readメソッドのジャバ言語コードの説明図である。

【図 13】

run()でのsync_readメソッド記述例を示す説明図である。

【図 14】

sync_burst_readのジャバ言語コードを示す説明図である。

【図 15】

endBurstAccessのジャバ言語コードを示す説明図である。

【図 16】

freeBurstBusLockのジャバ言語コードを示す説明図である。

【図 17】

run()でのバーストリードの記述例を示す説明図である。

【図 18】

sync_writeのジャバ言語コードを示す説明図である。

【図 19】

run()でのsync_writeメソッド記述例を示す説明図である。

【図 20】

sync_burst_writeのジャバ言語コードを示す説明図である。

【図 21】

run()でのバーストライトの記述例を示す説明図である。

【図 2 2】

ジャバ言語記述による実装例の概略仕様を示す説明図である。

【図 2 3】

コマンドインタフェースに関するrun() methodの実装例の一部を示す説明図である。

【図 2 4】

図 2 3 の実装例の続きを示す説明図である。

【図 2 5】

図 2 4 の実装例の続きを示す説明図である。

【図 2 6】

図 2 5 の実装例の続きを示す説明図である。

【図 2 7】

ユニファイドメモリに関するrun() methodの実装例を示す説明図である。

【図 2 8】

グラフィック描画ユニットに関するrun() methodの実装例を示す説明図である。

。

【図 2 9】

図 2 8 の実装例の続きを示す説明図である。

【図 3 0】

図 2 9 の実装例の続きを示す説明図である。

【図 3 1】

図 3 0 の実装例の続きを示す説明図である。

【図 3 2】

図 3 1 の実装例の続きを示す説明図である。

【図 3 3】

表示ユニットに関するrun() methodの実装例を示す説明図である。

【図 3 4】

図 3 3 の実装例の続きを示す説明図である。

【図 3 5】

中間表現への変換工程の詳細を全体的に例示する説明図である。

【図 3 6】

C-CFGの形式を例示する説明図である。

【図 3 7】

synchronizedの扱い（ハード合成用）について示す説明図である。

【図 3 8】

synchronizedの扱い（パラメトリック・モデルチェック用）について示す説明図である。

【図 3 9】

Command Interfaceの場合のCFGを示す説明図である。

【図 4 0】

Command Interfaceに関しハード合成用のCFGを例示する説明図である。

【図 4 1】

パラメトリック・モデルチェック用のCFGを例示する説明図である。

【図 4 2】

グラフィック描画ユニット（Graphics Rendering Unit）に関するCFGを例示する説明図である。

【図 4 3】

図 4 2 のCFGの続を示す説明図である。

【図 4 4】

パラメトリック・モデルチェック用のCFGを例示する説明図である。

【図 4 5】

図 4 4 のCFGの続を示す説明図である。

【図 4 6】

前記表示ユニット（Display Unit）に関するCFGを例示する説明図である。

【図 4 7】

パラメトリック・モデルチェック用のCFGを例示する説明図である。

【図 4 8】

コマンドインタフェース (Command Interface) の C F G、グラフィック描画ユニット (Graphics Rendering Unit) の C F G、及び前記表示ユニット (Display Unit) の C F Gにおける夫々の開始ノードの結合状態を例示する説明図である。

【図 4 9】

固定優先度スケジューラを例示する説明図である。

【図 5 0】

コマンドインタフェース (Command Interface) の C F Gに対する抽象化の様子を順を追って示す最初の説明図である。

【図 5 1】

図 5 0 の続を示す説明図である。

【図 5 2】

図 5 1 の続を示す説明図である。

【図 5 3】

図 5 2 の続を示す説明図である。

【図 5 4】

図 5 3 の続を示す説明図である。

【図 5 5】

図 5 4 の続を示す説明図である。

【図 5 6】

図 5 5 の続を示す説明図である。

【図 5 7】

グラフィック描画ユニット (Graphics Rendering Unit) の C F Gに対する抽象化処理の結果を例示する説明図である。

【図 5 8】

表示ユニット (Display Unit) の C F Gに対する抽象化処理の結果を例示する説明図である。

【図 5 9】

コマンドインタフェース (Command Interface) の C F Gに対する T N F A へ

の変換の様子を順を追って示す説明図である。

【図 6 0】

図 5 9 の続を示す説明図である。

【図 6 1】

図 6 0 の続を示す説明図である。

【図 6 2】

グラフィック描画ユニット (Graphics Rendering Unit) の C F G に対する T N F A への変換の様子を順を追って示す説明図である。

【図 6 3】

図 6 2 の続を示す説明図である。

【図 6 4】

図 6 3 の続を示す説明図である。

【図 6 5】

図 6 4 の続を示す説明図である。

【図 6 6】

表示ユニット (Display Unit) の C F G に対する T N F A への変換の様子を順を追って例示する説明図である。

【図 6 7】

図 6 6 の続を示す説明図である。

【図 6 8】

図 6 7 の続を示す説明図である。

【図 6 9】

図 6 1、図 6 4、図 6 8 で求めた T N F A の積 T N F A の構成例を示す説明図である。

【図 7 0】

パラメータに上限値を与えた時に、積 T N F A を構成する段階で制約を満たさない遷移枝の削除過程を示す説明図である。

【図 7 1】

図 7 0 の続を示す説明図である。

【図 7 2】

図 7 1 の続を示す説明図である。

【図 7 3】

図 7 2 の続を示す説明図である。

【図 7 4】

図 7 3 の続を示す説明図である。

【図 7 5】

図 7 4 の続を示す説明図である。

【図 7 6】

図 7 5 の続を示す説明図である。

【図 7 7】

C-TNFA2TNFAでAbstraction of TNFAを適時コールした場合の処理の経過の一部を示す説明図である。

【図 7 8】

図 7 7 の続を示す説明図である。

【図 7 9】

図 7 8 の続を示す説明図である。

【図 8 0】

得られたパラメータ条件に対して目的関数を各パラメータ値の合計として、これを最小化するという線形計画問題を解いた結果を例示する説明図である。

【図 8 1】

図 8 0 の結果に対し、 $kb1=2, kb2=1, kb3=1$ の制約を追加して得た解を例示する説明図である。

【図 8 2】

図 8 1 の結果に対し更に $kr1$ 以外のパラメータ変数を 1 以上という制約を追加して得た解を例示する説明図である。

【図 8 3】

BasicBlockへの実行サイクルの割り当てを例示する説明図である。

【図 8 4】

固定優先度スケジューラを例示する説明図である。

【図 8 5】

図 8 4 に示す固定優先度スケジューラの一部を用いて変形した後の C F G を例示する説明図である。

【図 8 6】

変形後の Command Interface の C F G の一部を例示する説明図である。

【図 8 7】

図 8 6 の続きを示す説明図である。

【図 8 8】

孤立ノードへの C F G の割り当てに関する C F G を例示する説明図である。

【図 8 9】

共有レジスタに関し C F G 生成のためのインライン展開の対象になる記述を例示する説明図である。

【図 9 0】

共有レジスタに関する擬似 C 記述を例示する説明図である。

【図 9 1】

図 9 0 の続を示す説明図である。

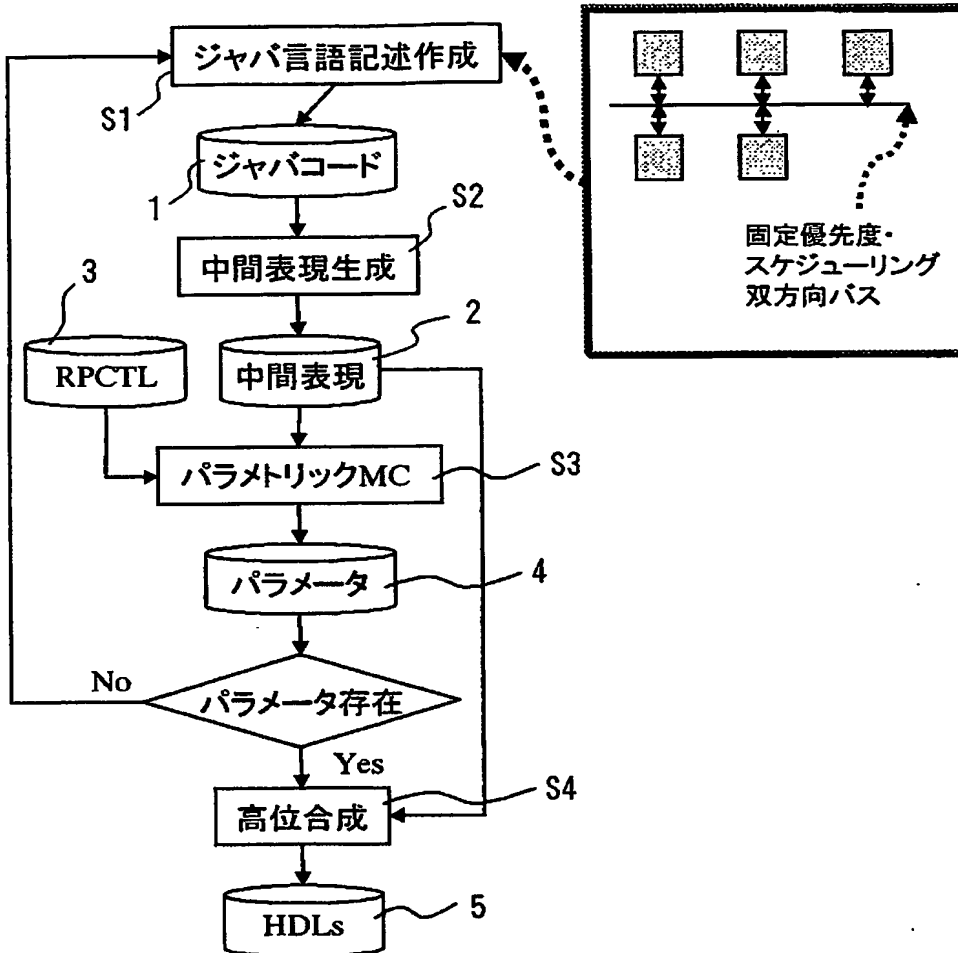
【符号の説明】

- 1 ジャバコード
- 2 中間表現
- 3 R P C T L
- 4 パラメータ
- 5 H D L による回路記述
- S 1 ジャバ言語記述によるモデル化
- S 2 中間表現生成
- S 3 パラメトリック・モデルチェッキング
- S 4 高位合成

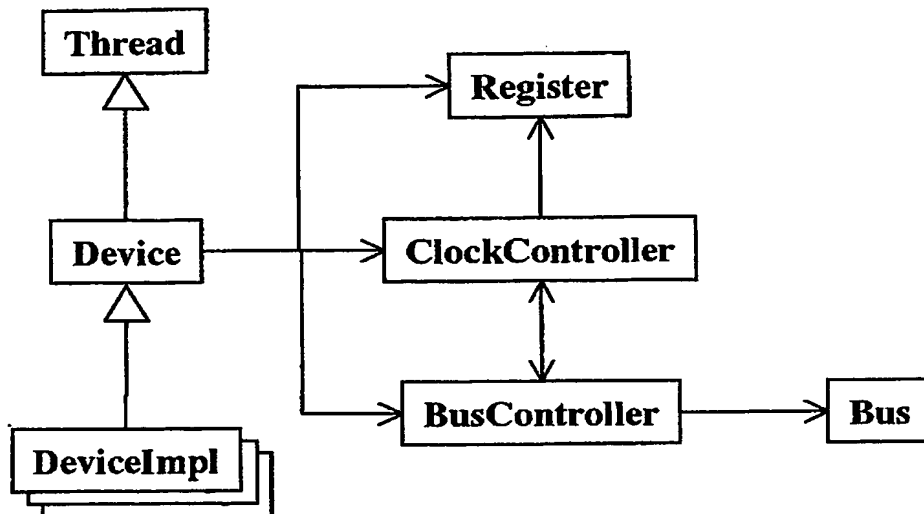
【書類名】 図面

【図 1】

図 1



【図 2】

図 2
〔モデル化(デザインパターン)〕

【図 3】

図 3
〔モデル化(クロック同期)〕

```

private void consume_1_clock() {
    /* 全デバイス数とこのmethodを実行したデバイスの数が等しいかチェック */
    if (++this.current_num == this.device_num) {
        this.current_num = 0;
        for (int i=0; i<this.device_num; i++) {
            /* レジスタ代入の実行 */
            registers[i].assignWriteValue();
        }
        /* バスロックを識別するフラグ変数(バスアクセス・フラグ)の初期化 */
        if (this.bc.getBusyCount() == 0) {
            this.bc.initLockDoneOnceFlag();
        }
        notifyAll();
    } else {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
}
}

```

【図 4】

図 4 〔モデル化(クロック同期)〕

```
public void assignWriteValue() {  
    /* 後述のsync_writeメソッド又は、sync_burst_writeメソッドにより  
       レジスタへの書き込みが実行がなされたかを判定 */  
    if (this.write_access) {  
        /* 実際に書き込みを行ったレジスタ(配列index)への書き込みを  
           実行 */  
        this.current_value[this.update_index] = this.write_value;  
        /* ライト・アクセス・フラグをリセット */  
        this.write_access = false;  
    }  
}
```


【図 5】

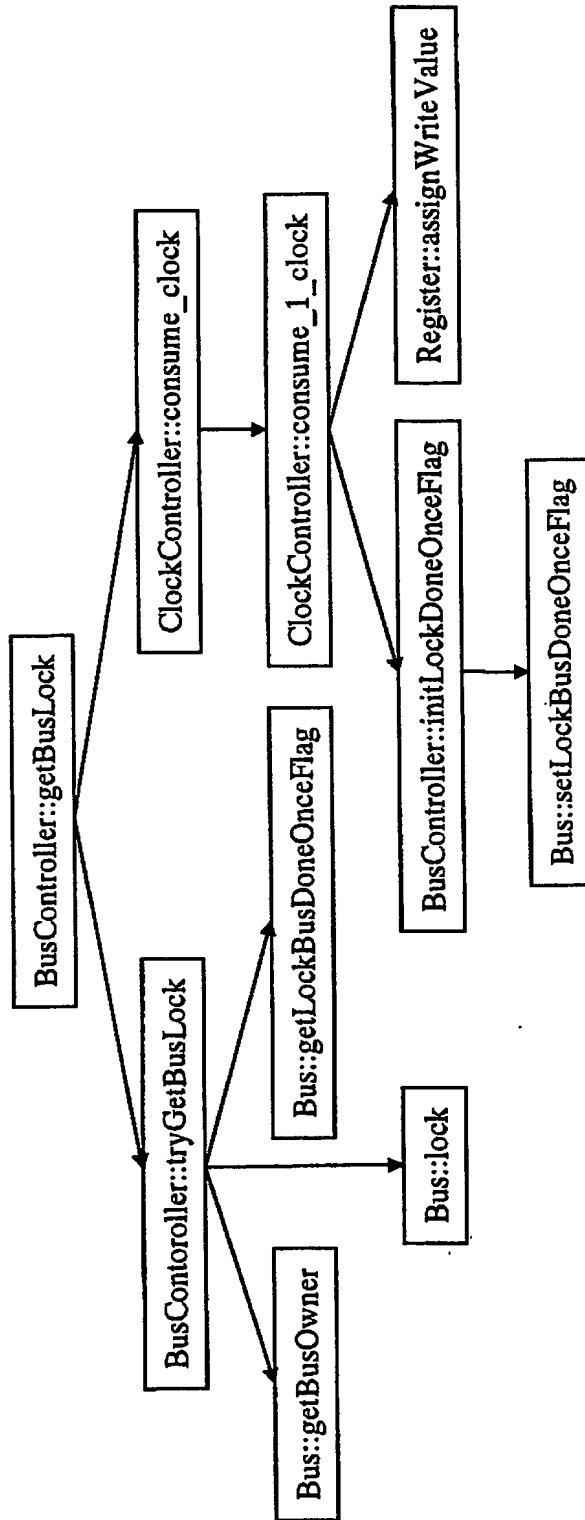
図 5
〔モデル化(バス権獲得)〕

method名	
getBusLock	tryGetBusLockがtrueを返せば、lockメソッドをコールする事でバス権を獲得し、falseの場合、consume_clockをコールし、clock_numの数だけクロック消費。
tryGetBusLock	getBusOwnerがnullで且つLockBusDoneOnceFlagがfalseなら、lockをコールし、getBusLockにtrueを返す。それ以外の場合はfalseを返す。
getBusOwner	現在バスをロックしているthreadのオブジェクトを返す。ロックしているオブジェクトが無ければnullを返す。
lock	現在、Bus classオブジェクト(lockメソッド)を実行しているThreadを、現在バスをロックしているオブジェクトとして登録する。
getBusDoneOnceFlag	バスアクセス・フラグの値を取得する。
consume_clock	consume_1_clockをコールする。
consume_1_clock	クロック同期メカニズムで既に説明済み。
initLockDoneOnceFlag	引数をfalseとしてsetLockBusDoneOnceFlagをコール。
setLockBusDoneOnceFlag	引数をバスアクセス・フラグに代入。
assignWriteValue	共有レジスタへのライトアクセスがあった場合、次のクロックへ進む前にレジスタ代入を実行する。

【図 6】

図 6

〔モデル化(バス権獲得)〕



【図 7】

図 7 〔モデル化(バス権獲得)〕

```
public void getBusLock(int clock_num) {  
    /* バス権が獲得出来たかチェック */  
    while (this.tryGetBusLock() == false) {  
        /* バス権が獲得出来なかったので、clock_numの数だけクロック消費 */  
        cc.consume_clock(clock_num);  
    }  
}
```

【図 8】

図 8 〔モデル化(バス権獲得)〕

```
private synchronized boolean tryGetBusLock() {  
    /* 現在バスがロックされておらず、且つそのクロック内で一度もバスが  
       ロックされた事がない(即ちバスアクセスフラグがfalse)か、をチェック */  
    if ((bus.getBusOwner() == null) && (bus.getLockDoneOnceFlag() == false)) {  
        /* バスをロック */  
        this.bus.lock();  
        /* ロック回数をインクリメント */  
        this.busycount++;  
        /* バス権が獲得できた事をtrueとして返す */  
        return true;  
    } else if (bus.getBusOwner() == Thread.currentThread()) {  
        /* 現在バスをロックしているスレッドによりさらにロック要求があった場合、  
           重ねてロックする。そのためにロック回数をインクリメント */  
        this.busycount++;  
        /* バス権が獲得できた事をtrueとして返す */  
        return true;  
    } else {  
        /* バス権が獲得出来なかった事をfalseとして返す */  
        return false;  
    }  
}
```

【図 9】

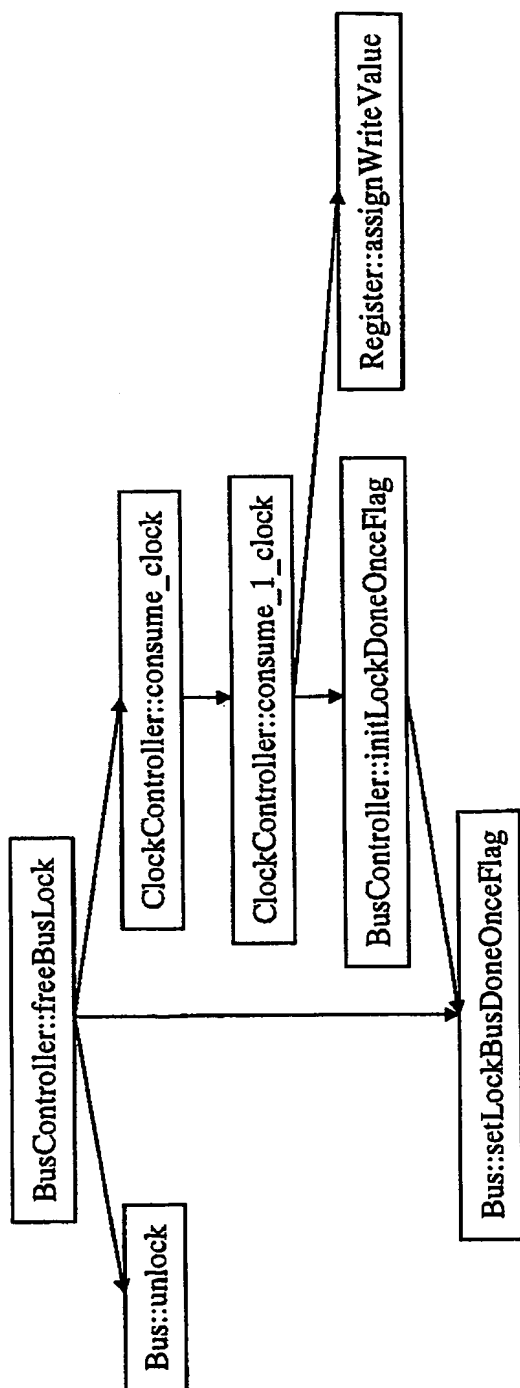
図 9
〔モデル化(バス権開放)〕

method名	
freeBusLock	unlockメソッドをコールする事でバス権を開放し、trueを引数として setLockBusDoneOnceFlagをコールし、consume_clockをコールし、 clock_numの数だけクロック消費。
unlock	現在バスをロックしているオブジェクトが、現在Bus classオブジェクト (lockメソッド)を実行しているThreadかを確認し、そうであれば、現在バスを ロックしているオブジェクトをnullとする。
consume_clock	consume_1_clockをコールする。
consume_1_clock	クロック同期メカニズムで既に説明済み。
initLockDoneOnceFlag	引数をfalseとしてsetLockBusDoneOnceFlagをコール。
setLockBusDoneOnceFlag	引数をバスアクセス・フラグに代入。
assignWriteValue	共有レジスタへのライトアクセスがあった場合、次のクロックへ進む前に レジスタ代入を実行する。

【図 10】

図 10

〔モデル化(バス権解放)〕



【図 1 1】

図 1 1 [モデル化(バス権解放)]

```
public void freeBusLock(int clock_num) {
    synchronized (this) {
        /* コールしたスレッドがバスをロックしているスレッドかチェック */
        if (this.bus.getBusOwner() == Thread.currentThread()) {
            /* ロック回数をデクリメント */
            this.busycount--;
            /* デクリメントした結果が0かどうかチェック */
            if (this.busycount == 0) {
                /* バスのロックを解放 */
                this.bus.unlock();
            }
            /* バスアクセス・フラグをtrueに設定 */
            this.bus.setLockDoneOnceFlag(true);
        }
    }
    /* clock_numの数だけクロック消費 */
    cc.consume_clock(clock_num);
}
```

【図 1 2】

図 1 2 [モデル化(排他的同期リード)]

```
public synchronized int sync_read(BusController bc,
                                   int index,
                                   int clock_num) {

    /* バス権獲得 */
    bc.getBusLock(clock_num);
    /* 指定したオブジェクト共有変数の値を読み込む */
    int read_value = this.current_value[index];
    /* バス権解放 */
    bc.freeBusLock(clock_num);
    /* 読み込んだ値を返す */
    return read_value;
}
```

[sync_read メソッド]

【図 13】

図 13 〔モデル化(排他的同期リード)〕

```
public void run() {  
    Register other_r0 = (Register)super.access_registers.get(0);  
    int read_value;  
    while (true) {  
        this.do_something_w_or_wo_clock_boundary1();  
        read_value = other_r0.sync_read(super.bc0, 1);  
        this.do_something_w_or_wo_clock_boundary2();  
    }  
}
```

〔run()での記述例〕

【図 14】

図 14 〔モデル化(排他的同期リード)〕

```
public int sync_burst_read(BusController bc, int index,  
                           int clock_num) {  
    /* コールされる度にロック条件を満たすならロックを重ねる */  
    bc.getBusLock(clock_num);  
    /* 指定したオブジェクト共有変数の値を読み込む */  
    int read_value = this.current_value[index];  
    /* 読み込んだ値を返す */  
    return read_value;  
}
```

〔sync_burst_read メソッド〕

【図 15】

図 15 〔モデル化(排他的同期リード)〕

```
public void endBurstAccess(BusController bc, int clock_num) {  
    bc.freeBurstBusLock(clock_num);  
}
```

〔endBurstAccess メソッド〕

【図 16】

図 16 〔モデル化(排他的同期リード)〕

```
public void freeBurstBusLock(int clock_num) {
    synchronized (this) {
        /* コールしたスレッドがバスをロックしているスレッドかチェック */
        if (this.bus.getBusOwner() == Thread.currentThread()) {
            /* ロック回数を0に戻す */
            this.busycount = 0;
            /* バスのロックを解放 */
            this.bus.unlock();
        }
        /* バスアクセス・フラグをtrueに設定 */
        this.bus.setLockDoneOnceFlag(true);
    }
    /* clock_numの数だけクロック消費 */
    cc.consume_clock(clock_num);
}
```

〔 freeBurstBusLock メソッド〕

【図 17】

図 17 〔モデル化(排他的同期リード)〕

```
public void run() {
    Register other_r1 = (Register)super.access_registers.get(1);
    int read_value[10]
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        /* バースト・リード(10回連続リード) */
        synchronized (this) {
            int i;
            for (i=0; i<10; i++) {
                read_value[i] = other_r1.sync_burst_read(super.bc, i, 1);
                super.cc.consume_clock(1);
                this.do_something_wo_clock_boundary1();
            }
            other_r1.sync_burst_read(super.bc, i, 1);
            other_r1.endBurstAccess(super.bc, 1);
            this.do_something_wo_clock_boundary2();
        }
        this.do_something_w_or_wo_clock_boundary2();
    }
}
```

〔run()での記述例〕

【図 18】

図 18 [モデル化(排他的同期ライト)]

```

public synchronized void sync_write(BusController bc,
                                     int write_value,
                                     int index, int clock_num) {

    /* バス権獲得 */
    bc.getBusLock(clock_num);
    /* 指定したオブジェクト共有変数への書き込み値を保持 */
    this.write_value = write_value;
    /* アクセスした配列を通知 */
    this.update_index = index;
    /* 共有変数への書き込みがあった事を通知
       (consume_1_clockにて次クロックへの遷移
       直前に共有変数に書き込みを実施) */
    this.write_access = true;
    /* バス権解放 */
    bc.freeBusLock(clock_num);
}

```

[sync_write メソッド]

【図 19】

図 19 [モデル化(排他的同期ライト)]

```

public void run() {
    Register other_r0 = (Register)super.access_registers.get(0);
    int write_value;
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        other_r0.sync_write(super.bc, write_value0, 1);
        this.do_something_w_or_wo_clock_boundary2();
    }
}

```

[run()での記述例]

【図 20】

図 20

〔モデル化(排他的同期ライント)〕

```
public int sync_burst_read {(BusController bc, int write_value, int index, int clock_num) {  
    /* コールされる度にロック条件を満たすならロックを重ねる */  
    bc.getBusLock(clock_num);  
    /* 指定したオブジェクト共有変数への書き込み値を保持 */  
    this.write_value = write_value;  
    /* 共有変数への書き込みがあった事を通知  
       (consume_1_clockにて次クロックへの遷移  
       直前に共有変数に書き込みを実施) */  
    this.write_access = true;  
}
```

sync_burst_write メソッド

【図 2 1】

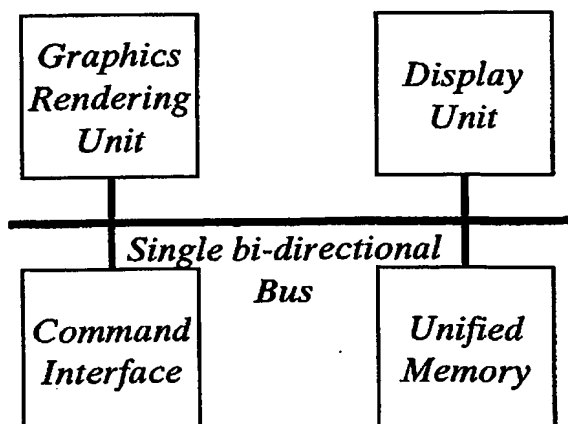
図 2 1 〔モデル化(排他的同期ライト)〕

```
public void run() {  
    Register other_r0 = (Register)super.access_registers.get(0);  
    int write_value[10]  
    while (true) {  
        this.do_something_w_or_wo_clock_boundary1();  
        /* バースト・ライト(10回連続ライト) */  
        synchronized (this) {  
            int i;  
            for (i=0; i<10; i++) {  
                other_r0.sync_burst_write(super.bc, write_value[i], 1));  
                super.cc.consume_clock(1);  
                this.do_something_wo_clock_boundary1();  
            }  
            other_r1.sync_burst_write(super.bc, write_value[i], 1);  
            other_r1.endBurstAccess(super.bc, 1);  
            this.do_something_wo_clock_boundary2();  
        }  
        this.do_something_w_or_wo_clock_boundary2();  
    }  
}
```

run()での記述例

【図 2 2】

図 2 2



【図 23】

図 23

[run() methodの実装例(Command Interface)]

```

public void run() {
    Register mem_con_reg = (Register)super.access_registers.get(2);
    int[] drawing_commands = {100, 101, 102, 103, 104};
    int dcom_index = 0;
    int com_flag_0 = 0;
    int com_flag_1 = 0;
    while (true) {
        /* 外部入力信号write_req信号があり、且つwait_flag信号がfalseのとき
        コマンド受け付けを実行。それ以外の時は、単にクロックを消費するだけ。*/
        if (this.getWriteReqSignal() && (this.wait_flag == false)) {
            /* コマンドの受付を実行。*/
            this.input_command = drawing_commands[dcom_index++];
            /* 一定サイクル消費。ここでは3クロックとした。*/
            super.cc.consume_clock(3);
            /* wait_flag変数をtrueにセット*/
            this.wait_flag = true;
            /* write_req信号がありwait_flag変数がtrueなので、
            wait出力信号をtrueにセットし、1クロック消費。*/
            this.wait_output_signal = true;
            super.cc.consume_clock(1);
        }
    }
}

```

シミュレーション用に
挿入した記述

【図 24】

図 24
[run() methodの実装例(Command Interface)]

```
/* コマンドフラグの値が0の場合に対応する描画コマンドの値を更新する。  
0,1共にコマンドフラグの値が0の場合は、0に対応するほうを優先する。  
0,1共にコマンドフラグの値が1の場合は、一定クロック待機して再度実行。*/  
while (true) {  
    synchronized (this) {  
        /* コマンドフラグ0,1をパーストリードする。*/  
        com_flag_0 = mem_con_reg.sync_burst_read(super.bc, 0, 1);  
        super.cc.consume_clock(1); //クロック消費。  
        com_flag_1 = mem_con_reg.sync_burst_read(super.bc, 2, 1);  
        super.cc.consume_clock(1); //クロック消費。  
        if (com_flag_0 == 0) {  
            /* コマンドフラグ0の値が0の場合。または、コマンドフラグ0,1の値が共に0の場合。*/  
            /* 描画コマンド0に入力されたコマンドを書き込む。*/  
            mem_con_reg.sync_burst_write(super.bc, input_command, 1, 1);  
            super.cc.consume_clock(1);  
            /* コマンドフラグ0の値を1にする。*/  
            mem_con_reg.sync_burst_write(super.bc, 1, 0, 1);  
            /* バーストモードの終了(クロック消費含む)。*/  
            mem_con_reg.endBurstAccess(super.bc, 1);  
            break;  
        }  
    }  
}
```

【図 25】

図 25

〔 run() methodの実装例 (Command Interface) 〕

```
} else if (com_flag_1 == 0) {  
    /* コマンドフラグ1の値が0の場合。*/  
    /* 描画コマンド1に入力されたコマンドを書き込む。*/  
    mem_con_reg.sync_burst_write(super.bc, input_command, 3, 1);  
    super.cc.consume_clock(1);  
    /* コマンドフラグ1の値を1にする。*/  
    mem_con_reg.sync_burst_write(super.bc, 1, 2, 1);  
    /* パースモードの終了(クロック消費含む)。*/  
    mem_con_reg.endBurstAccess(super.bc, 1);  
    break;  
}  
} else {  
    /* 0,1共にコマンドフラグの値が1の場合。*/  
    /* パースモードの終了(クロック消費含む)。*/  
    mem_con_reg.endBurstAccess(super.bc, 1);  
    /* 一定クロック待機。ここでは3クロック待機させた。*/  
    super.cc.consume_clock(3);  
}  
} // end of synchronized  
} // end of nested while-loop
```

【図 26】

図 26

[run() methodの実装例(Command Interface)]

```
/* wait_flag変数をfalseにセット。 */  
this.wait_flag = false;  
/* wait_flag変数がfalseになったので、wait出力信号をfalseに戻し、1クロック消費。 */  
this.wait_output_signal = false;  
super.cc.consume_clock(1);  
/* 描画コマンド配列を最後まで使用した場合はそのインデックスを最初に戻す。 */  
if (dcom_index == drawing_commands.length) {  
    dcom_index = 0;  
}  
} else {  
    super.cc.consume_clock(1);  
}  
} // end of while-loop  
}
```

シミュレーション用に
挿入した記述

【図 27】

図 27 [run() methodの実装例 (Unified Memory)]

```
public void run() {  
    /* Unified Memoryへの書き込みを行うバス上のデバイス内の  
       レジスタをインスタンス化 */  
    // Graphics Rendering Unit 内のレジスタ群  
    Register renderer_reg = (Register)super.access_registers.get(0);  
    // Display Unit 内のレジスタ群  
    Register display_reg = (Register)super.access_registers.get(1);  
    // Command Interface 内のレジスタ群  
    Register com_fetch_reg = (Register)super.access_registers.get(2);  
    /* モデル簡略化の為、実際には何も行わない */  
    while (true) {  
        /* 1クロック消費。 */  
        super.cc.consume_clock(1);  
    }  
}
```

【図 28】

図 28 [run() methodの実装例 (Graphics Rendering Unit)]

```
public void run() {  
    Register mem_con_reg = (Register)super.access_registers.get(2);  
    int[] rendering_result = new int[6];  
    int current_command = 0;  
    int read_data = 0;  
    int com_flag_0 = 0;  
    int com_flag_1 = 0;  
    while (true) {  
        /* 待ち状態。 */  
        super.cc.consume_clock(3);  
        while (true) {  
            synchronized (this) {  
                /* コマンドフラグ0、1をバーストリードする。 */  
                com_flag_0 = mem_con_reg.sync_burst_read(super.bc, 0, 1);  
                super.cc.consume_clock(1); //クロック消費。  
                com_flag_1 = mem_con_reg.sync_burst_read(super.bc, 2, 1);  
                /* バーストモードの終了(クロック消費含む)。 */  
                mem_con_reg.endBurstAccess(super.bc, 1);  
            } // end of synchronized  
        }  
    }  
}
```

【図 29】

図 29

[run() methodの実装例(Graphics Rendering Unit)]

```

if (com_flag_0 == 1) {
    /* コマンドフラグ0の値が0の場合。または、コマンドフラグ0,1の値が共に0の場合。*/
    synchronized(this) {
        /* 描画コマンド0の値を読み込む。*/
        current_command = mem_con_reg.sync_burst_read(super.bc, 1, 1);
        /* render_startをtrueに。*/
        this.render_start = true;
        super.cc.consume_clock(1);
        for (int i=0; i<3; i++) {
            /* データの読み出し。*/
            read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
            /* クロック消費。*/
            super.cc.consume_clock(1);
            /* レンダリング。*/
            rendering_result[i] = this.rendering(read_data, current_command);
        } // end of for-loop
        /* コマンドフラグ0の値を0にする。*/
        mem_con_reg.sync_burst_write(super.bc, 0, 0, 1);
        /* バーストモードの終了(クロック消費含む)。*/
        mem_con_reg.endBurstAccess(super.bc, 1);
    } // end of synchronized
}

```

【図 30】

図 30

〔 run() methodの実装例 (Graphics Rendering Unit) 〕

```
for (int i=3; i<6; i++) {  
    /* レンダリング。*/  
    rendering_result[i] = this.rendering(read_data, current_command);  
    /* クロック消費。*/  
    super.cc.consume_clock(1);  
} // end of for-loop  
break;  
} else if (com_flag_1 == 1) {  
    /* コマンドフラグ1の値が0の場合。*/  
    synchronized(this) {  
        /* 描画コマンド1の値を読み込む。*/  
        current_command = mem_con_reg.sync_burst_read(super.bc, 3, 1);  
        /* render_startをtrueに。*/  
        this.render_start = true;  
        super.cc.consume_clock(1);  
        for (int i=0; i<3; i++) {  
            /* データの読み出し。*/  
            read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);  
            /* クロック消費。*/  
            super.cc.consume_clock(1);  
            /* レンダリング。*/  
            rendering_result[i] = this.rendering(read_data, current_command);  
        } // end of for-loop
```

【図 3 1】

図 3 1 [run() methodの実装例 (Graphics Rendering Unit)]

```

/* コマンドフラグ1の値を0にする。*/
mem_con_reg.sync_burst_write(super.bc, 0, 2, 1);
/* バーストモードの終了(クロック消費含む)。*/
mem_con_reg.endBurstAccess(super.bc, 1);
} // end of synchronized
for (int i=3; i<6; i++) {
    /* レンダリング。*/
    rendering_result[i] = this.rendering(read_data, current_command);
    /* クロック消費。*/
    super.cc.consume_clock(1);
} // end of for-loop
break;
} else {
    /* 0,1共にコマンドフラグの値が1の場合。*/
    /* 一定クロック待機。ここでは3クロック待機させた。*/
    super.cc.consume_clock(3);
}
} // end of nested while-loop

```

【図 3 2】

図 3 2 [run() methodの実装例 (Graphics Rendering Unit)]

```

/* レンダリングされたデータをメモリに書き込む。*/
synchronized (this) {
    for (int i=0; i<6; i++) {
        if (i == 5) {
            /* データの書き込み。*/
            mem_con_reg.sync_burst_write(super.bc, rendering_result[i], i+4, 1);
            /* バーストモードの終了(クロック消費含む)。*/
            mem_con_reg.endBurstAccess(super.bc, 1);
        } else {
            /* データの書き込み。*/
            mem_con_reg.sync_burst_write(super.bc, rendering_result[i], i+4, 1);
            /* クロック消費。*/
            super.cc.consume_clock(1);
        }
    } // end of for-loop
} // end of synchronized
/* レンダリング終了。*/
this.render_start = false;
} // end of while-loop
}

```

【図 3 3】

図 3 3 [run() methodの実装例 (Display Unit)]

```
public void run() {
    Register mem_con_reg = (Register)super.access_registers.get(2);
    int read_data = 0;
    while (true) {
        synchronized (this) {
            for (int i=0; i<6; i++) {
                if (i == 5) {
                    /* データの読み出し。*/
                    read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
                    /* バーストモードの終了(クロック消費含む)。*/
                    mem_con_reg.endBurstAccess(super.bc, 1);
                } else {
                    /* データの読み出し。*/
                    read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
                    /* 表示の為のデータロード開始。*/
                    this.display_start = true;
                    /* クロック消費。*/
                    super.cc.consume_clock(1);
                }
            } // end of for-loop
        } // end of synchronized
    }
}
```

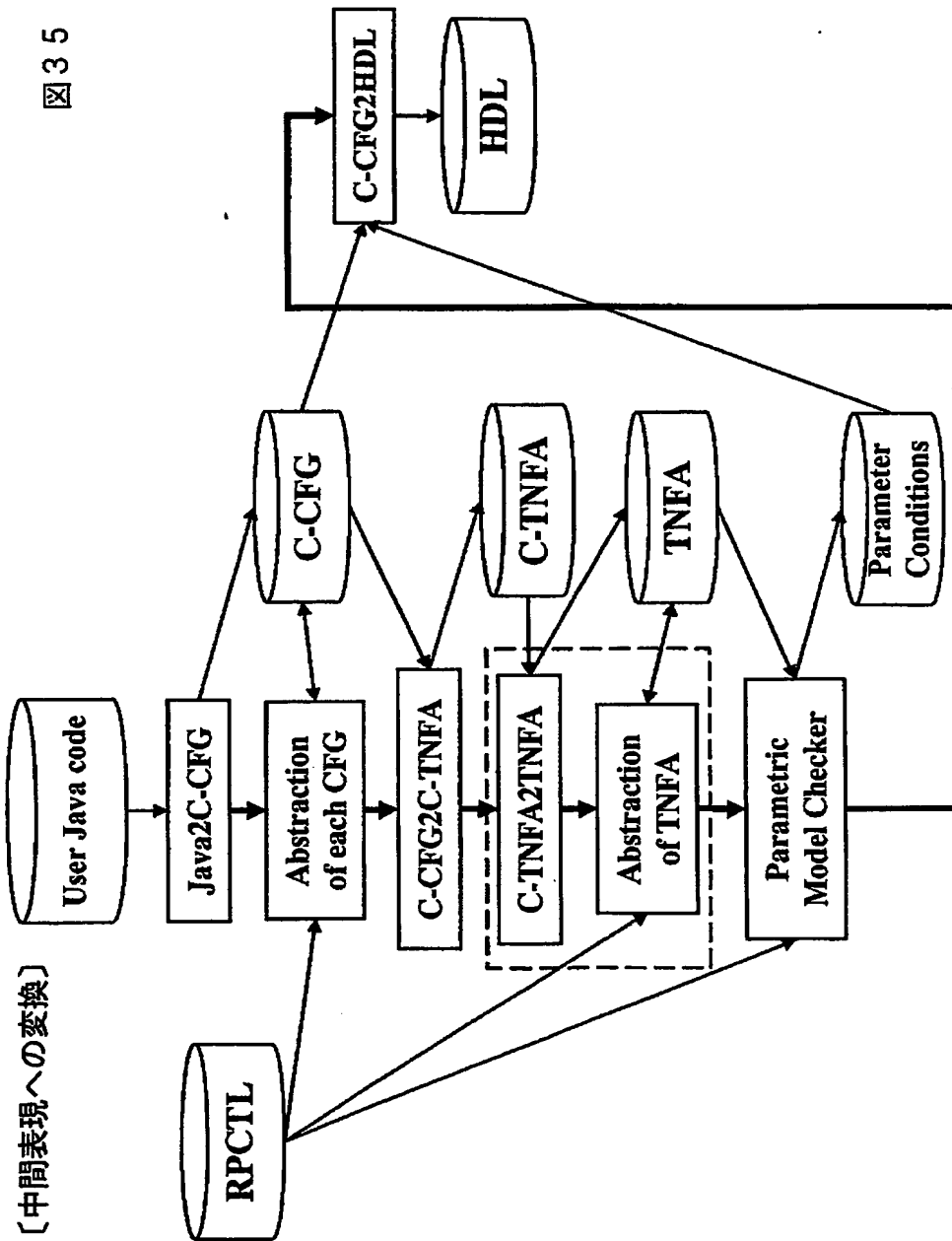
【図 3 4】

図 3 4 [run() methodの実装例 (Display Unit)]

```
/* 表示の為のデータロード終了。*/
this.display_start = false;
/* 表示。*/
for (int i=0; i<6; i++) {
    this.display(read_data);
}
/* 待ち。*/
super.cc.consume_clock(3);
}
}
```

【図 35】

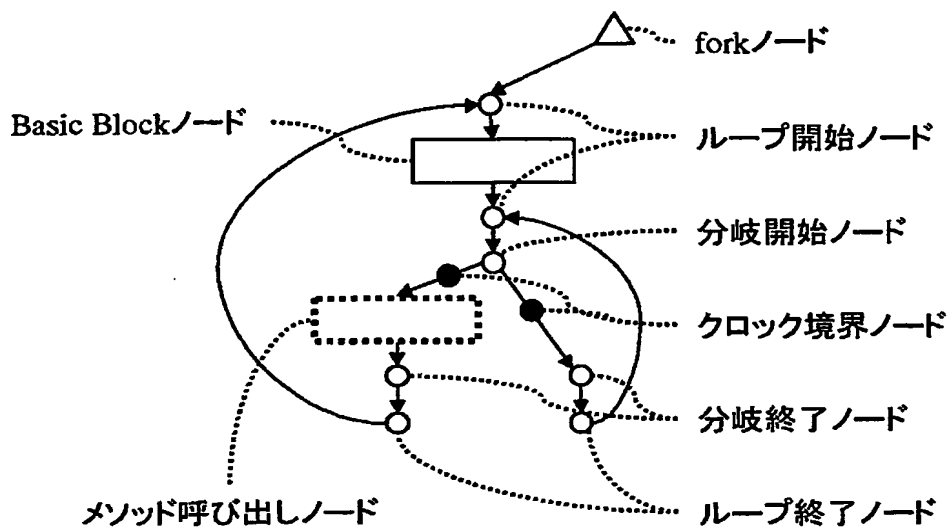
図 35



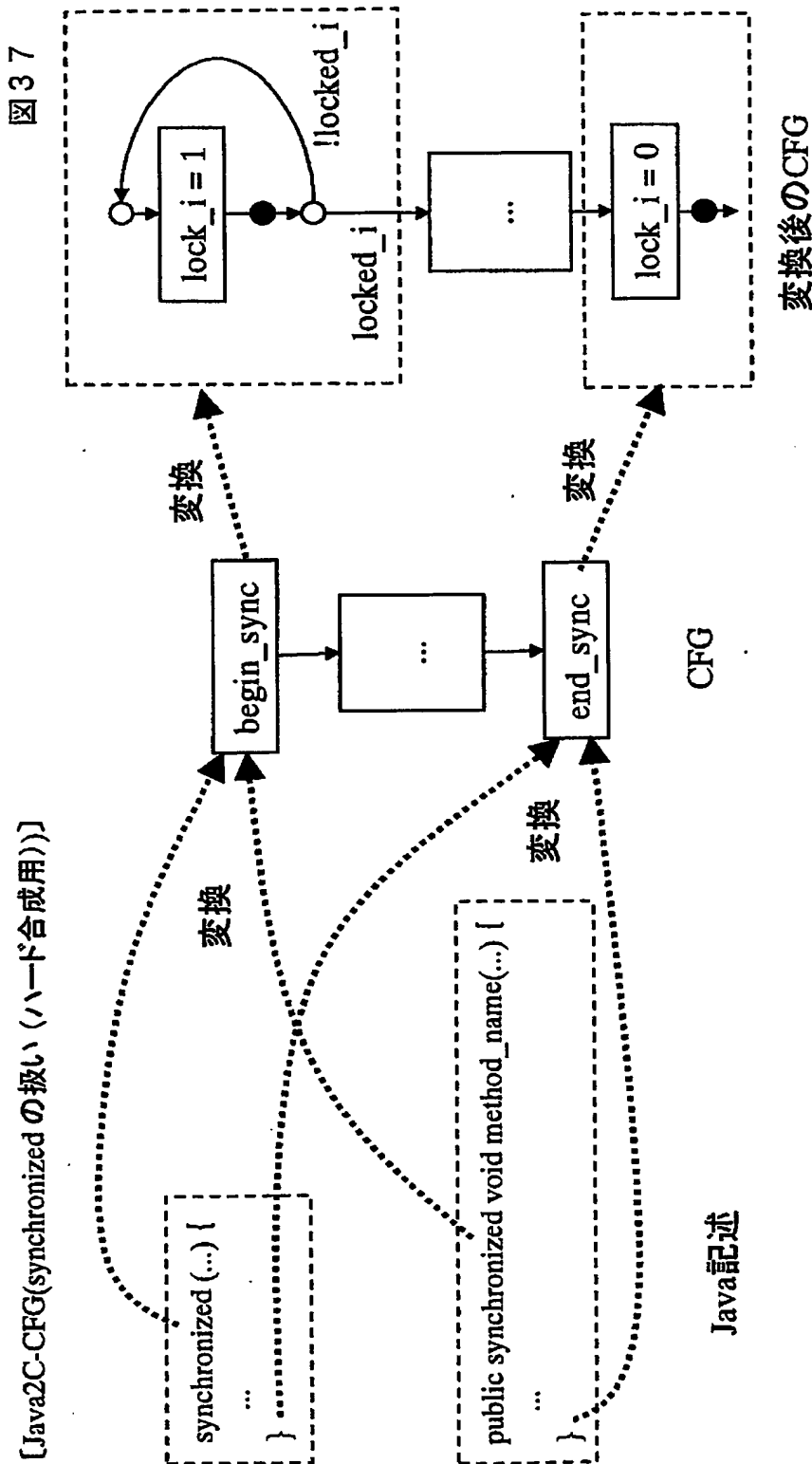
【図 3 6】

図 3 6

[Java2C-CFG(C-CFGの形式)]



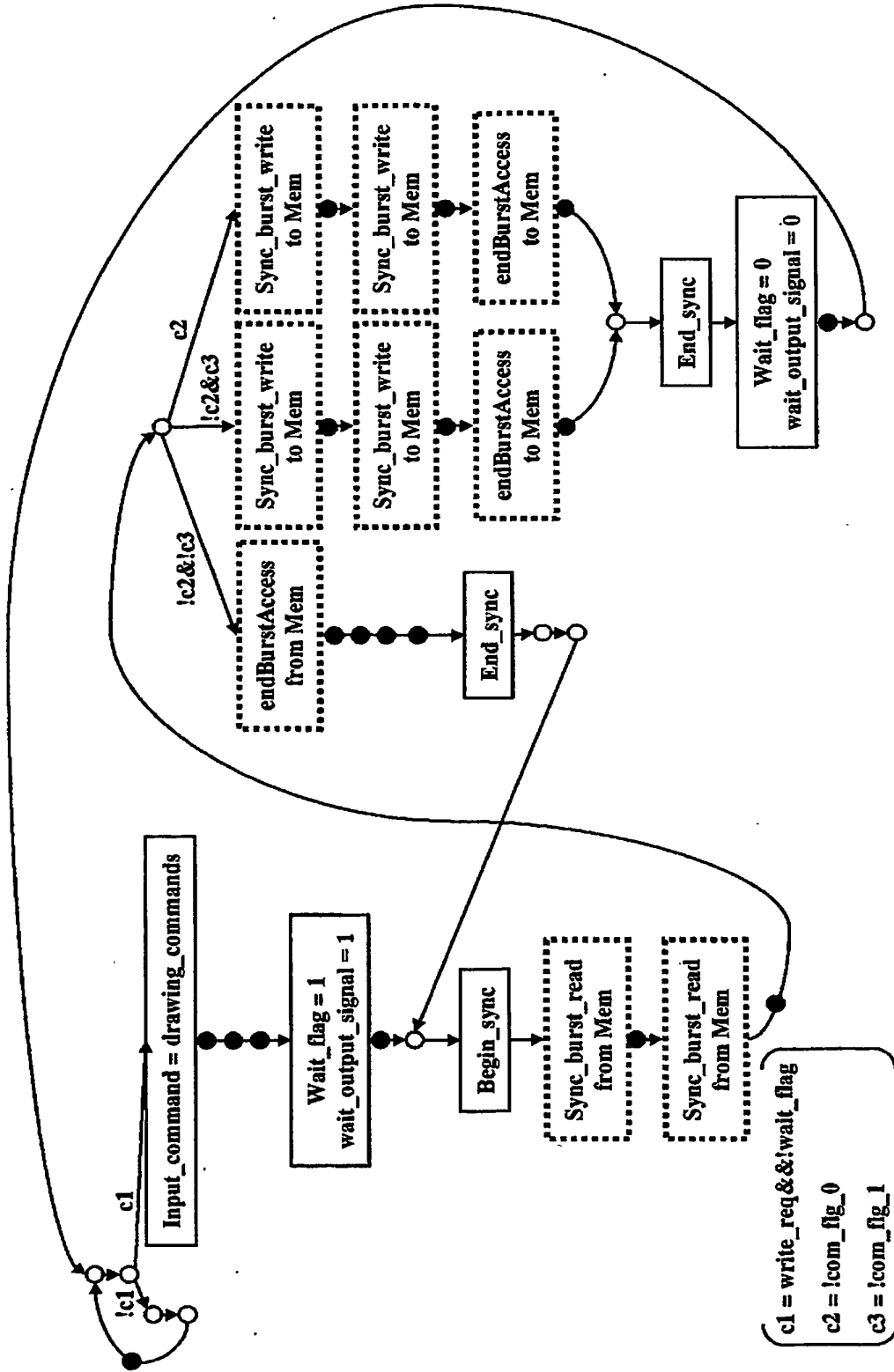
【図 37】



【図 39】

図 39

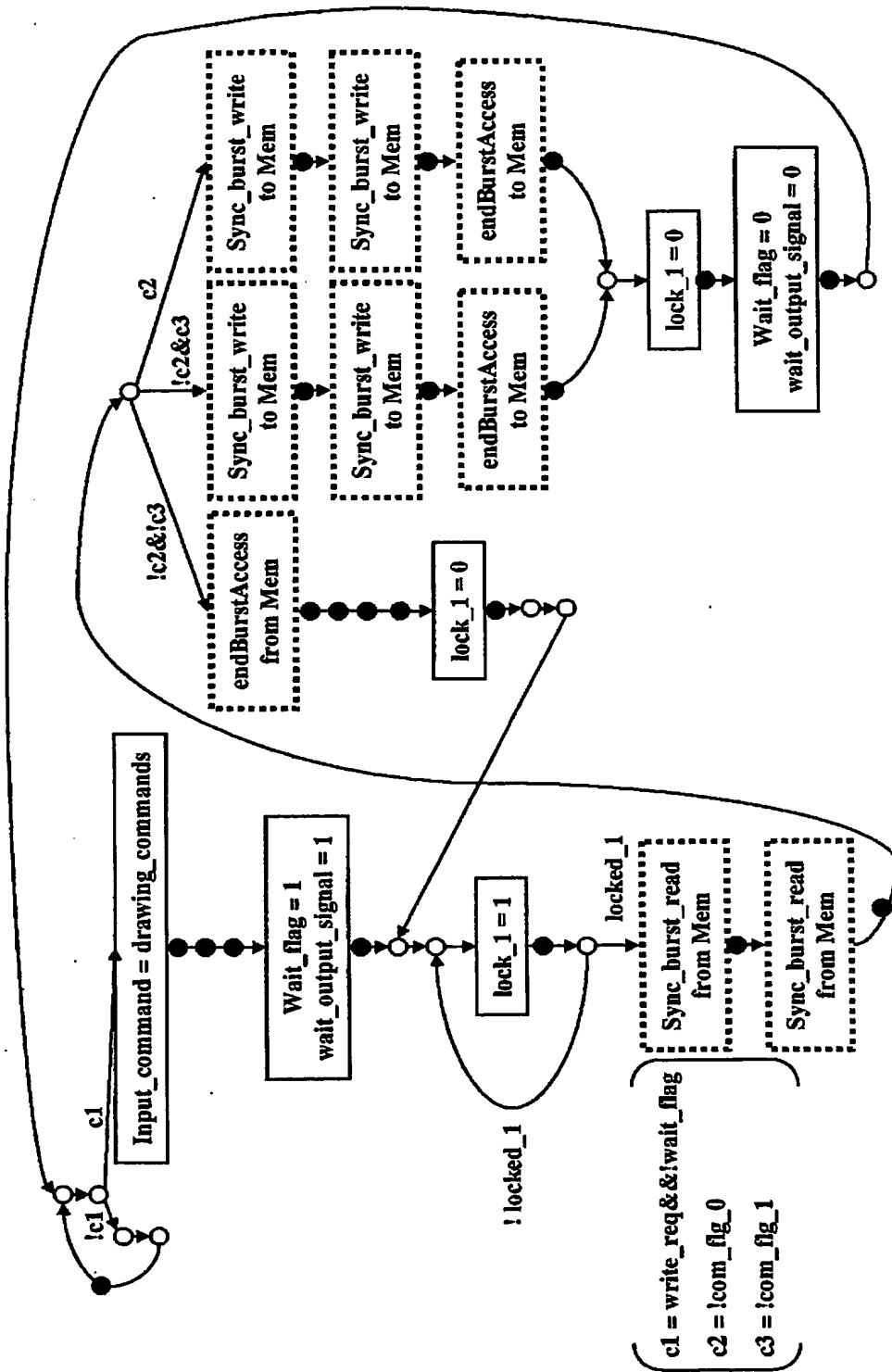
【Java2C-CFG(Command Interface (CFG))】



【図 40】

図 40

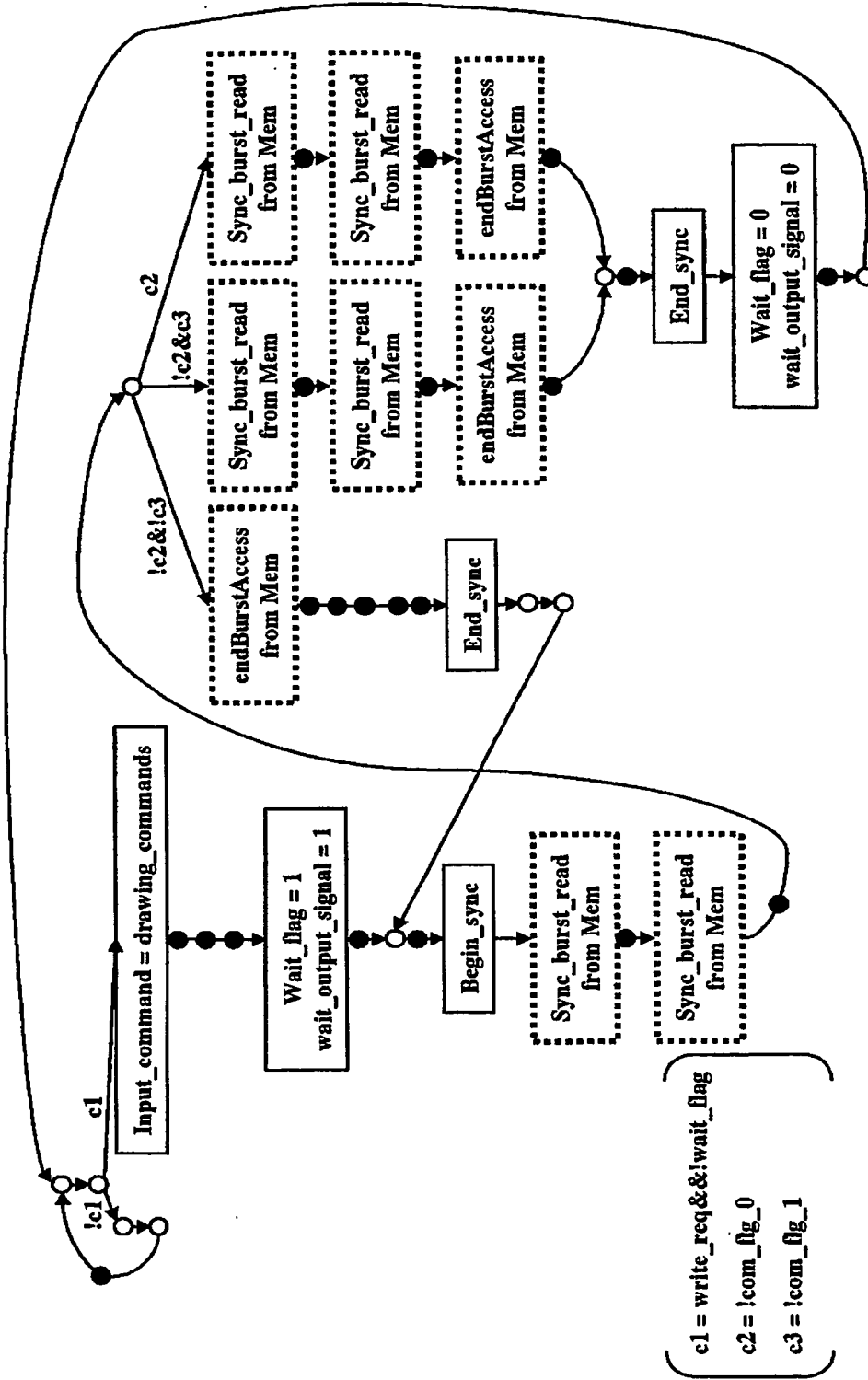
【Java2C-CFG (Command Interface (ハード合成用))】



【図 4 1】

図 4 1

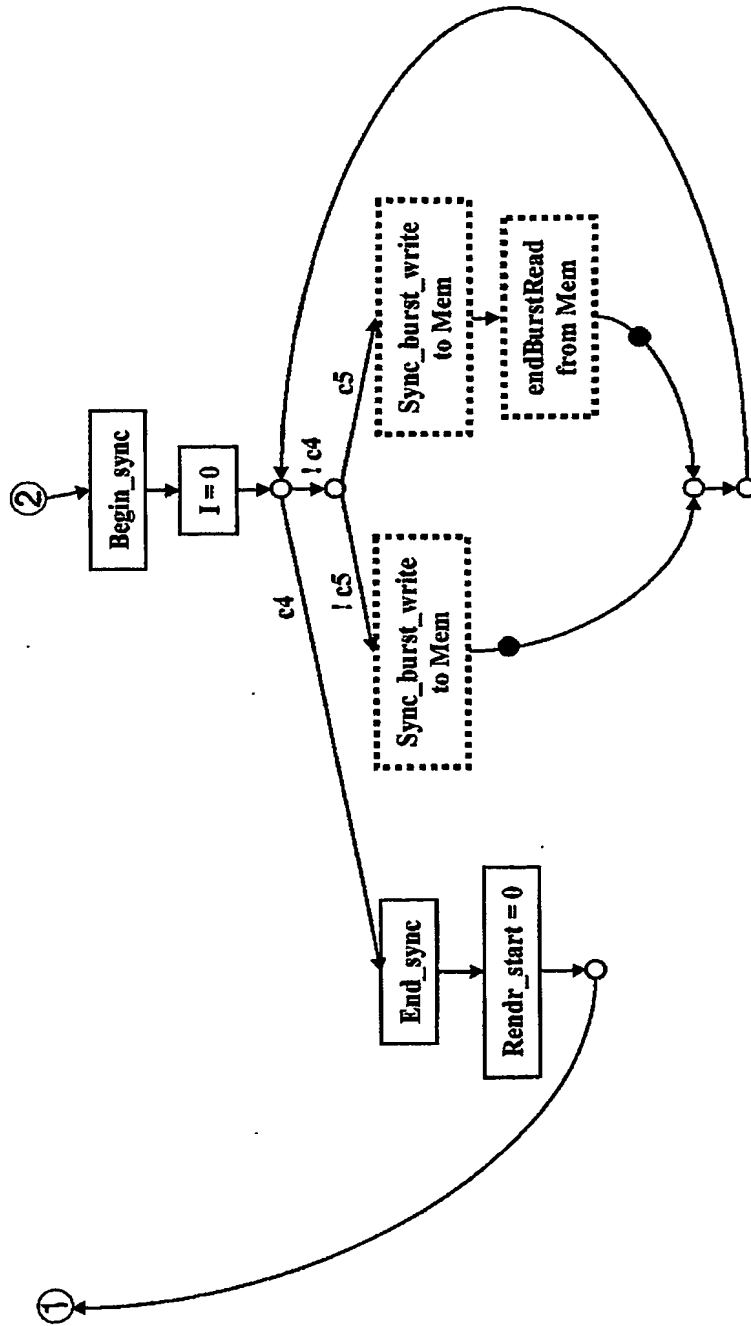
【Java2C-CFG (Command Interface (パラメトリック・モデルチェッキング用))】



【図 4 3】

図 4 3

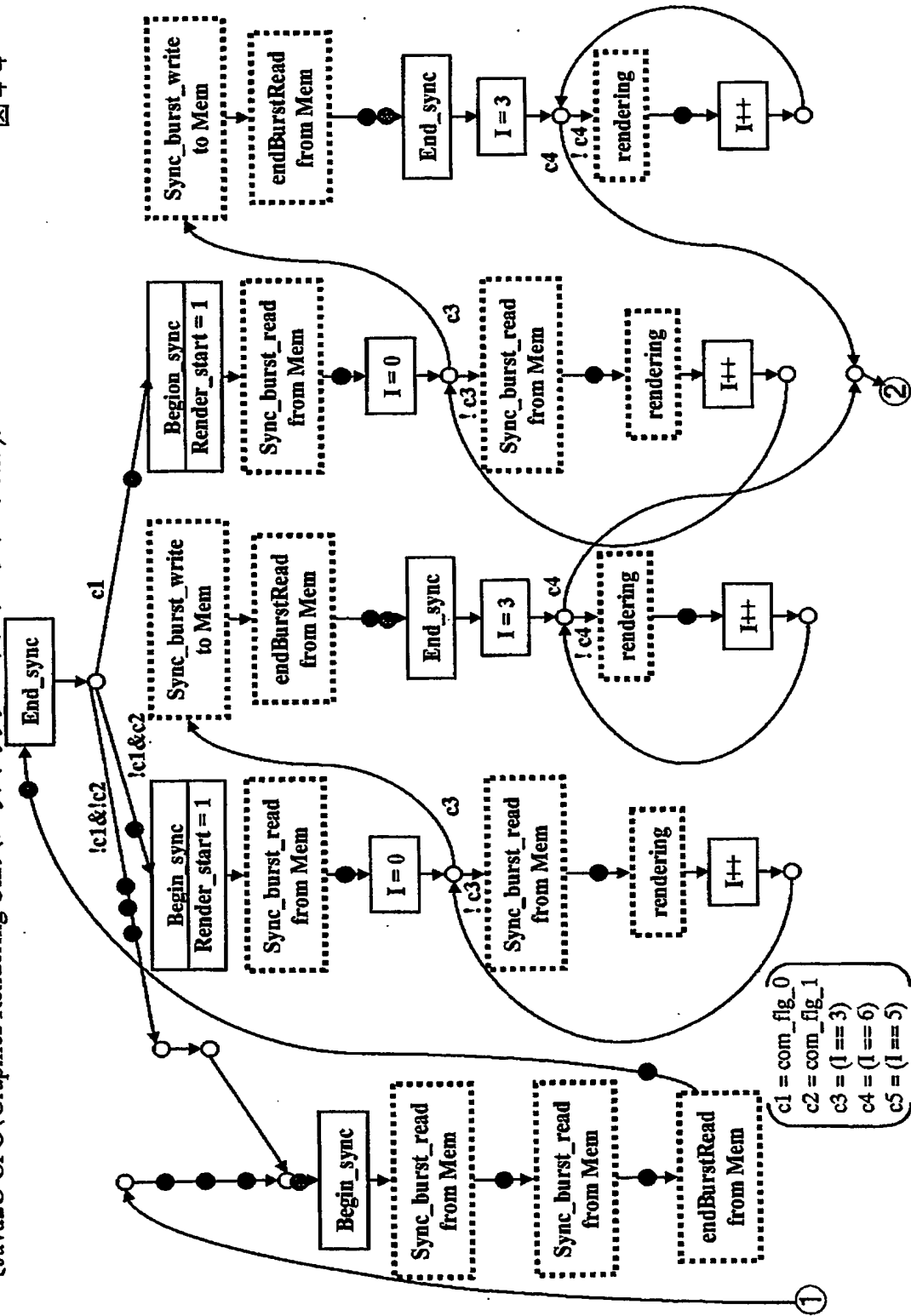
【Java2C-CFG (Graphics Rendering Unit (CFG 続き))】



【図44】

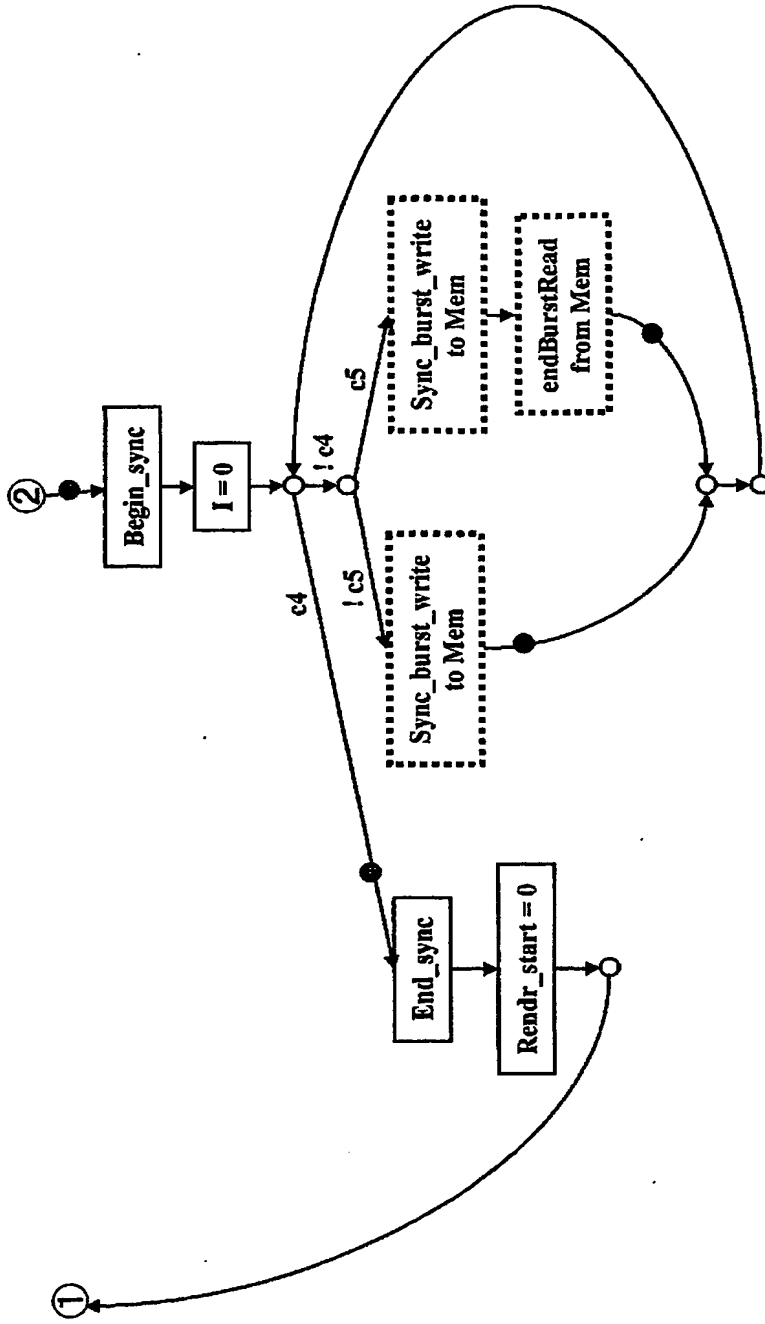
図44

【Java2C-CFG (Graphics Rendering Unit (パラメトリック・モデルチェッキング用))】

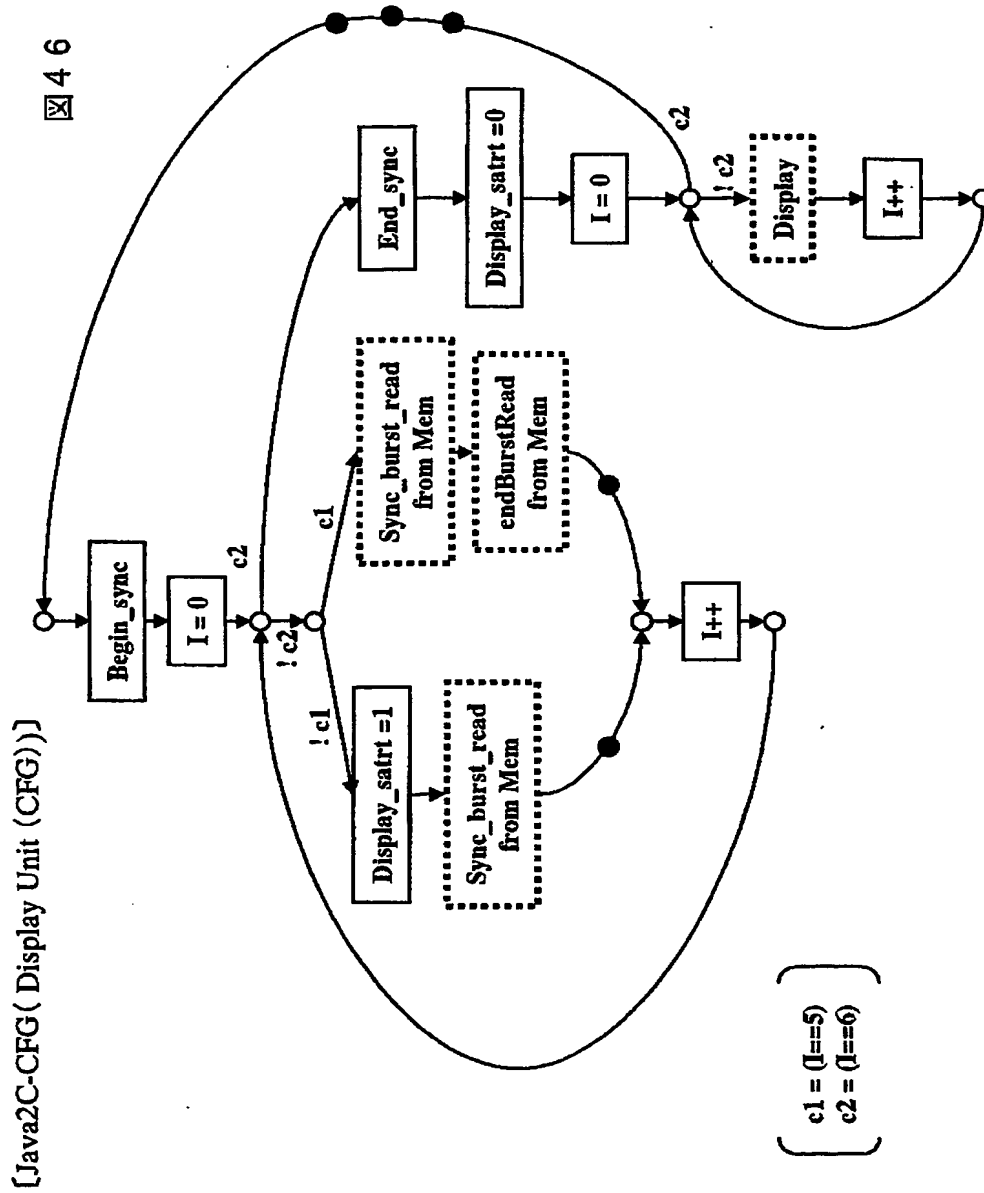


【図 4 5】

図 4 5
[Java2C-CFG(Graphics Rendering Unit (パラメトリック・モデルチェッキング用続き))]



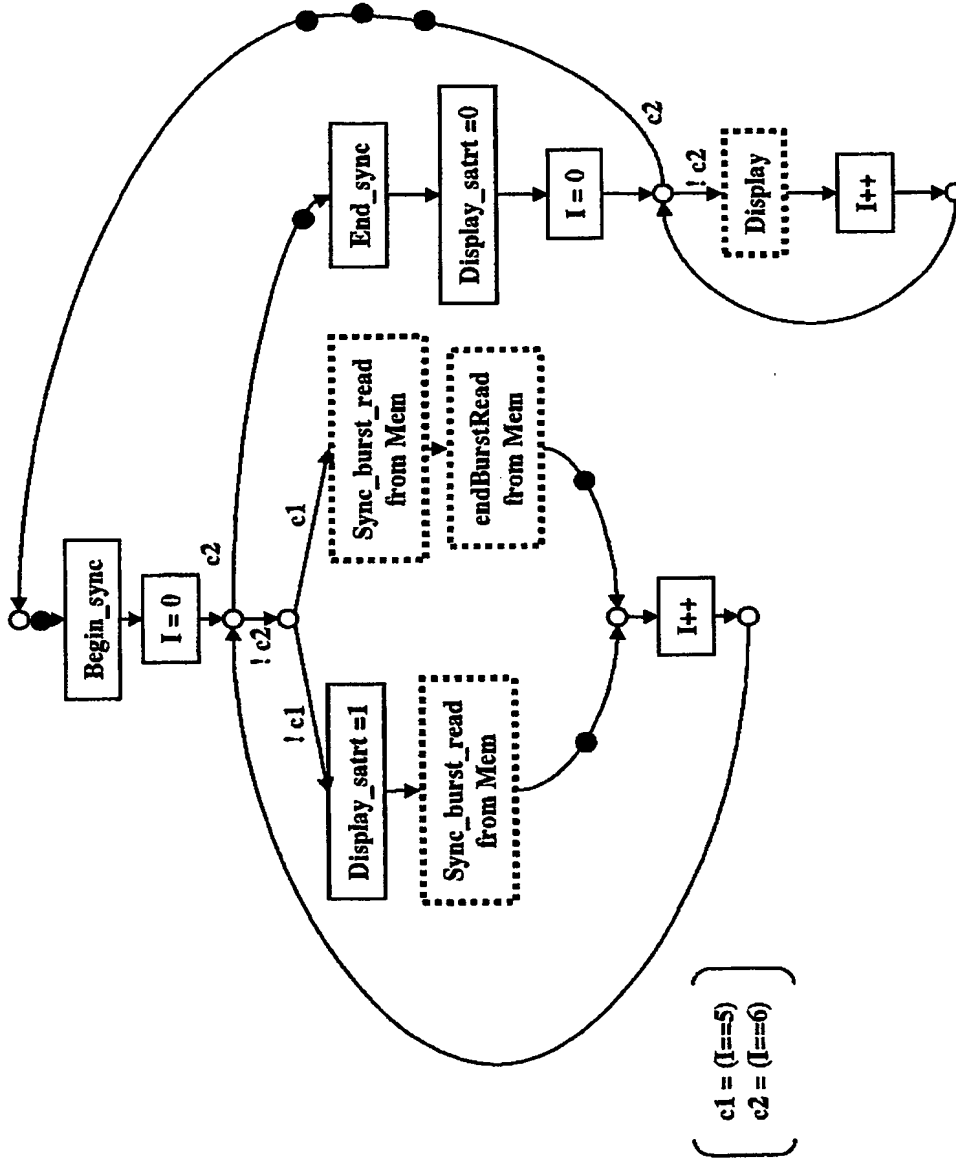
【図 46】



【図 47】

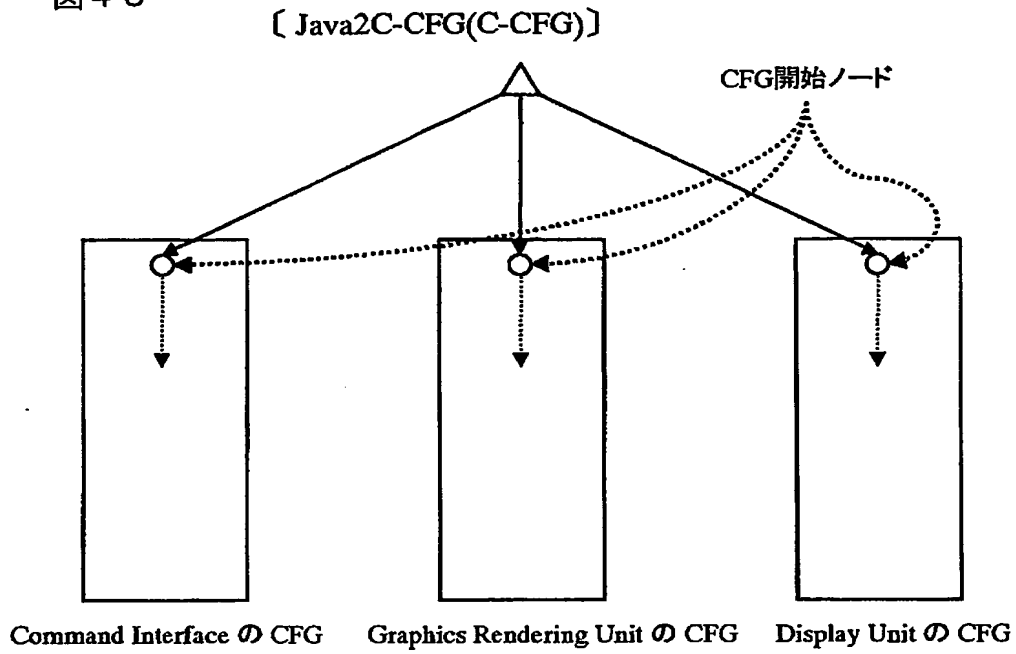
図 47

【Java2C-CFG (Display Unit (パラメトリック・モデルチェッキング用))】



【図 48】

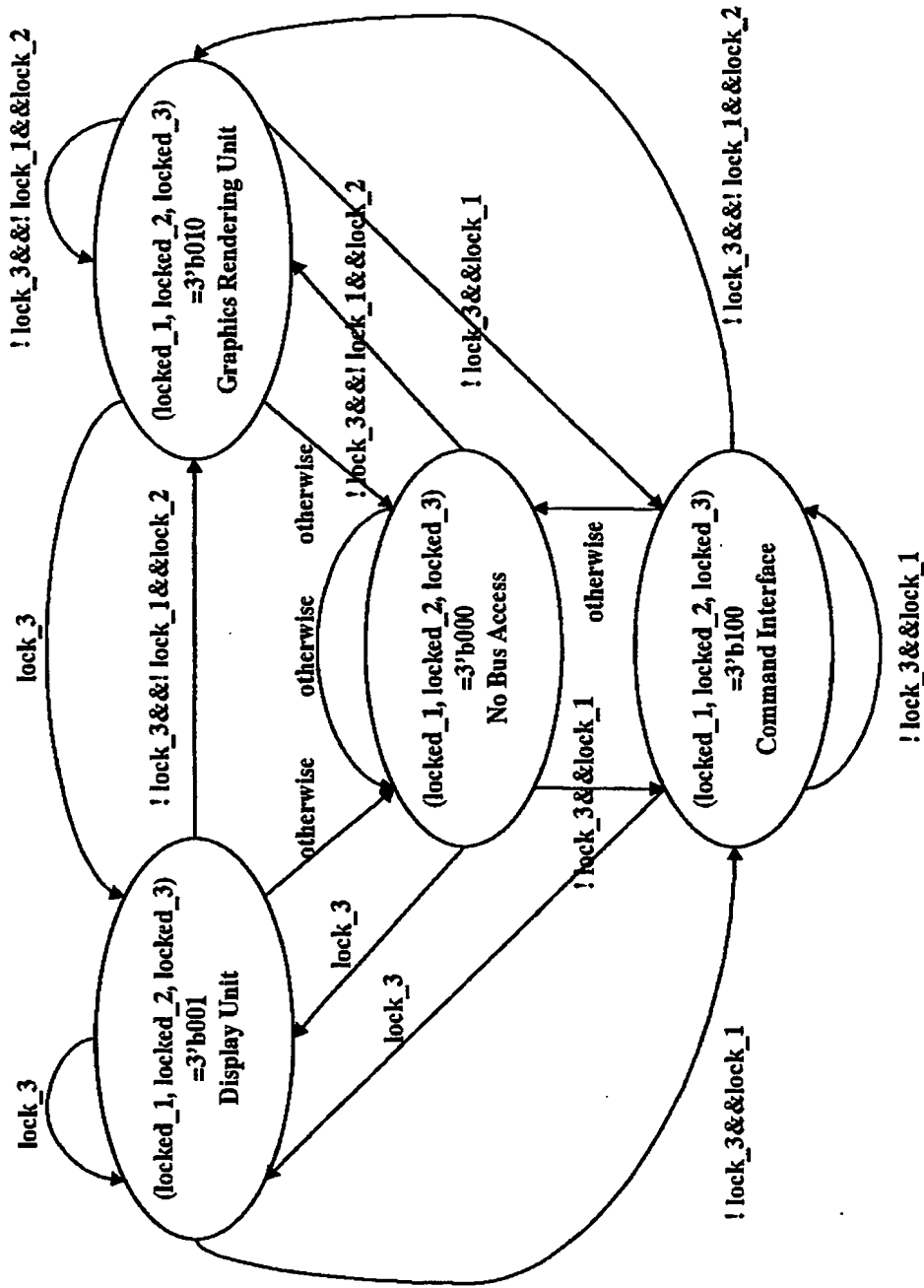
図 48



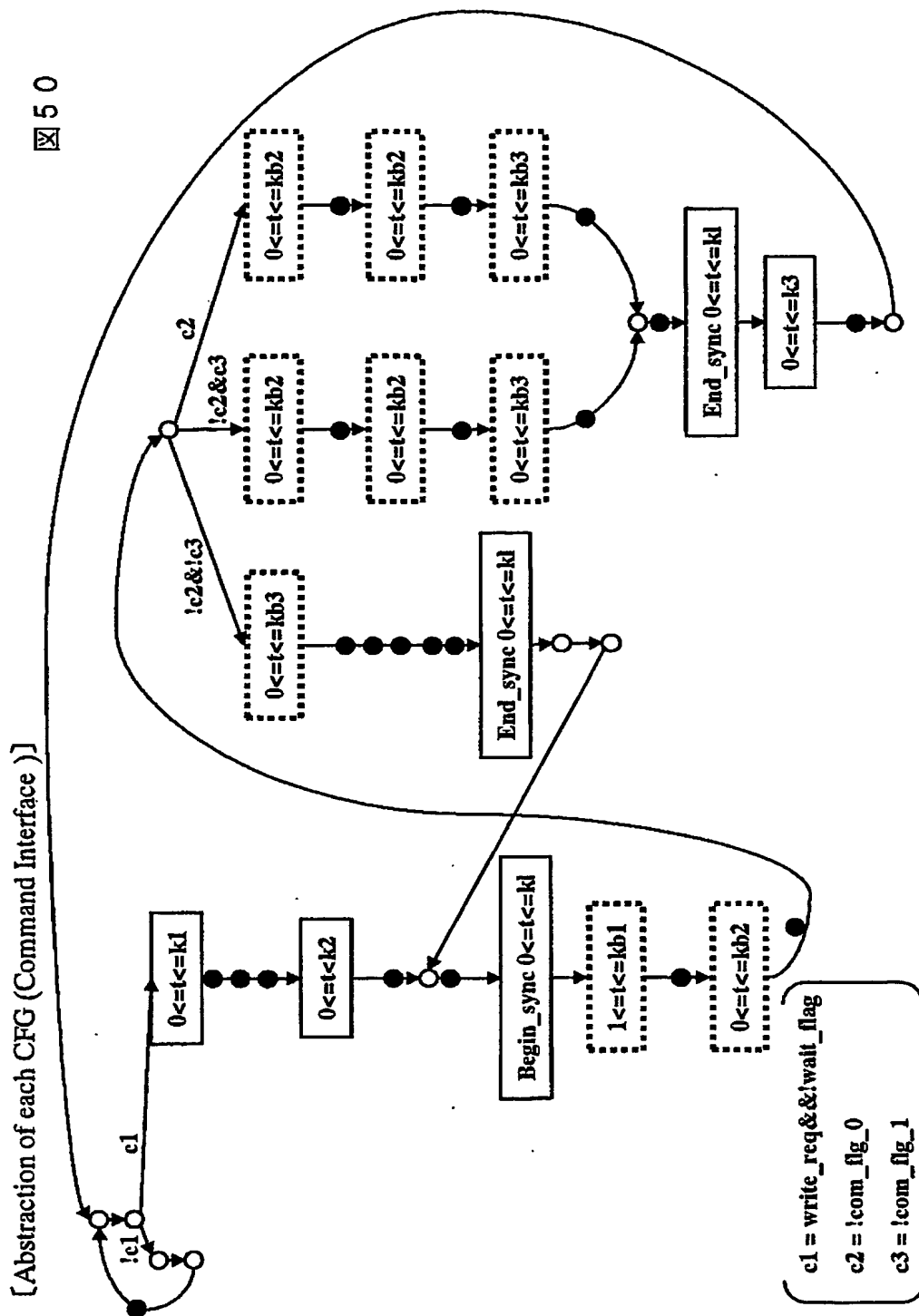
【図 49】

図 49

[Java2C-CFG(Fixed Priority Scheduler)]



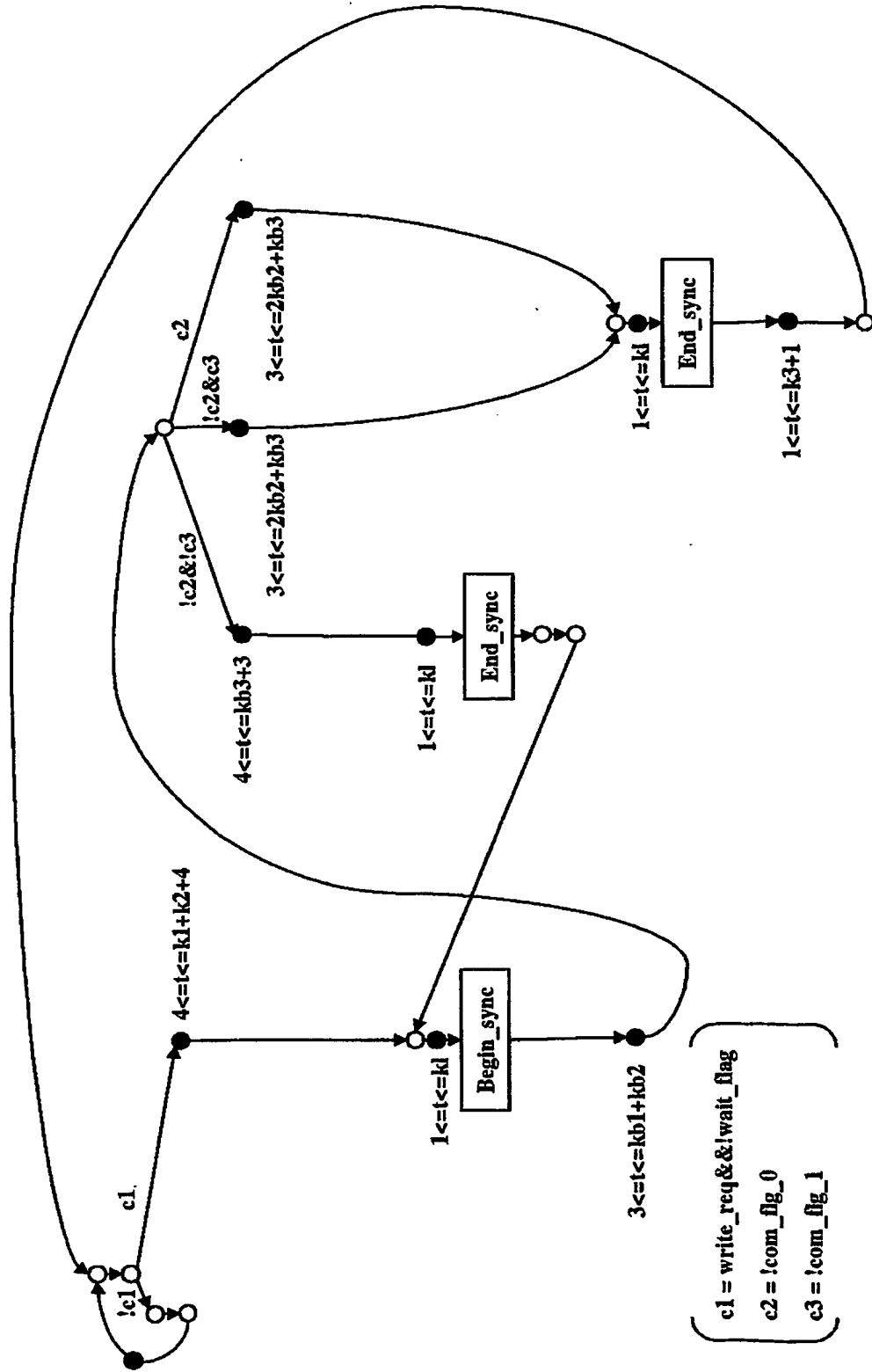
【図 50】



【図 51】

図 51

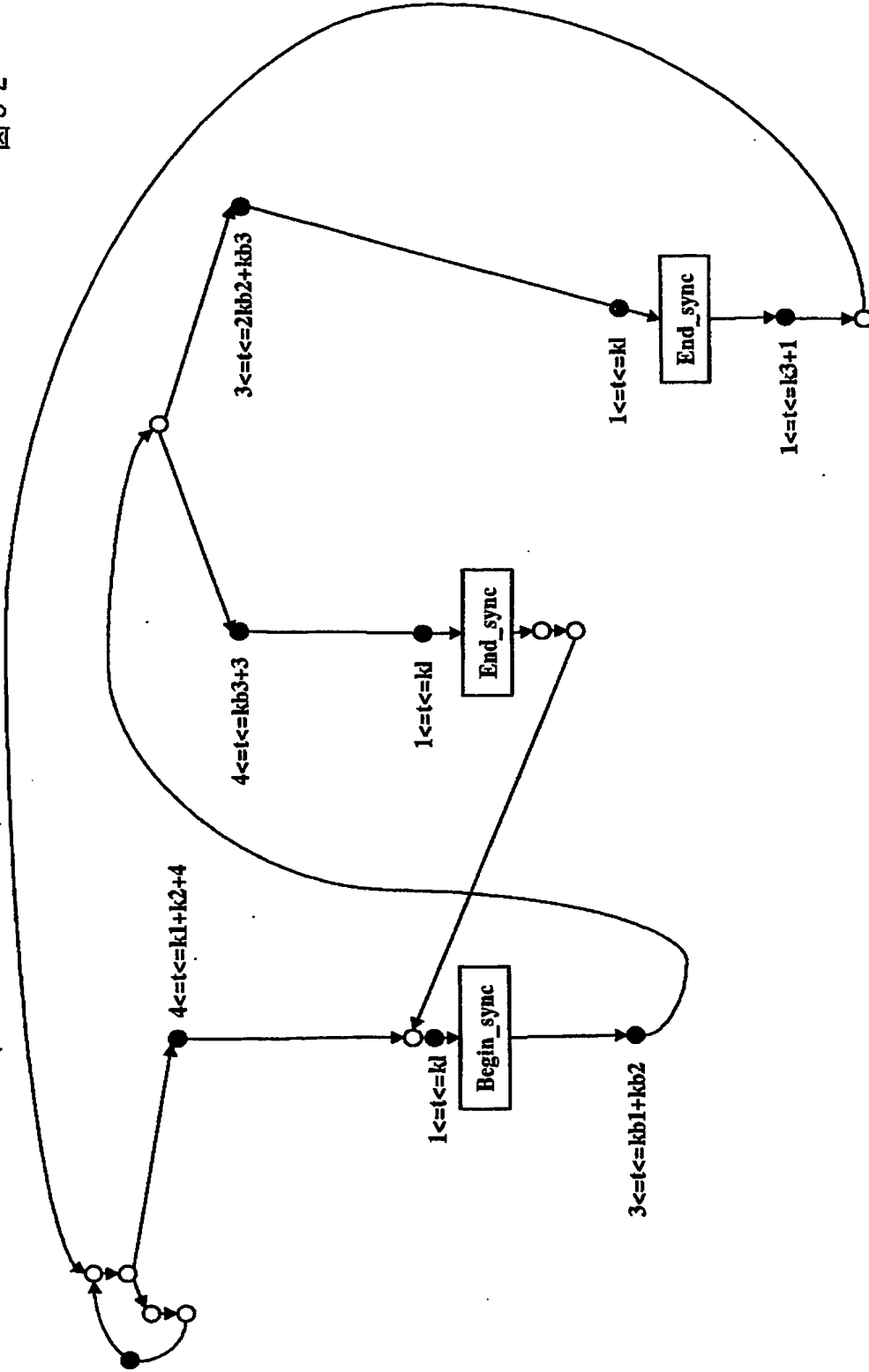
[Abstraction of each CFG (Command Interface)]



【図 5 2】

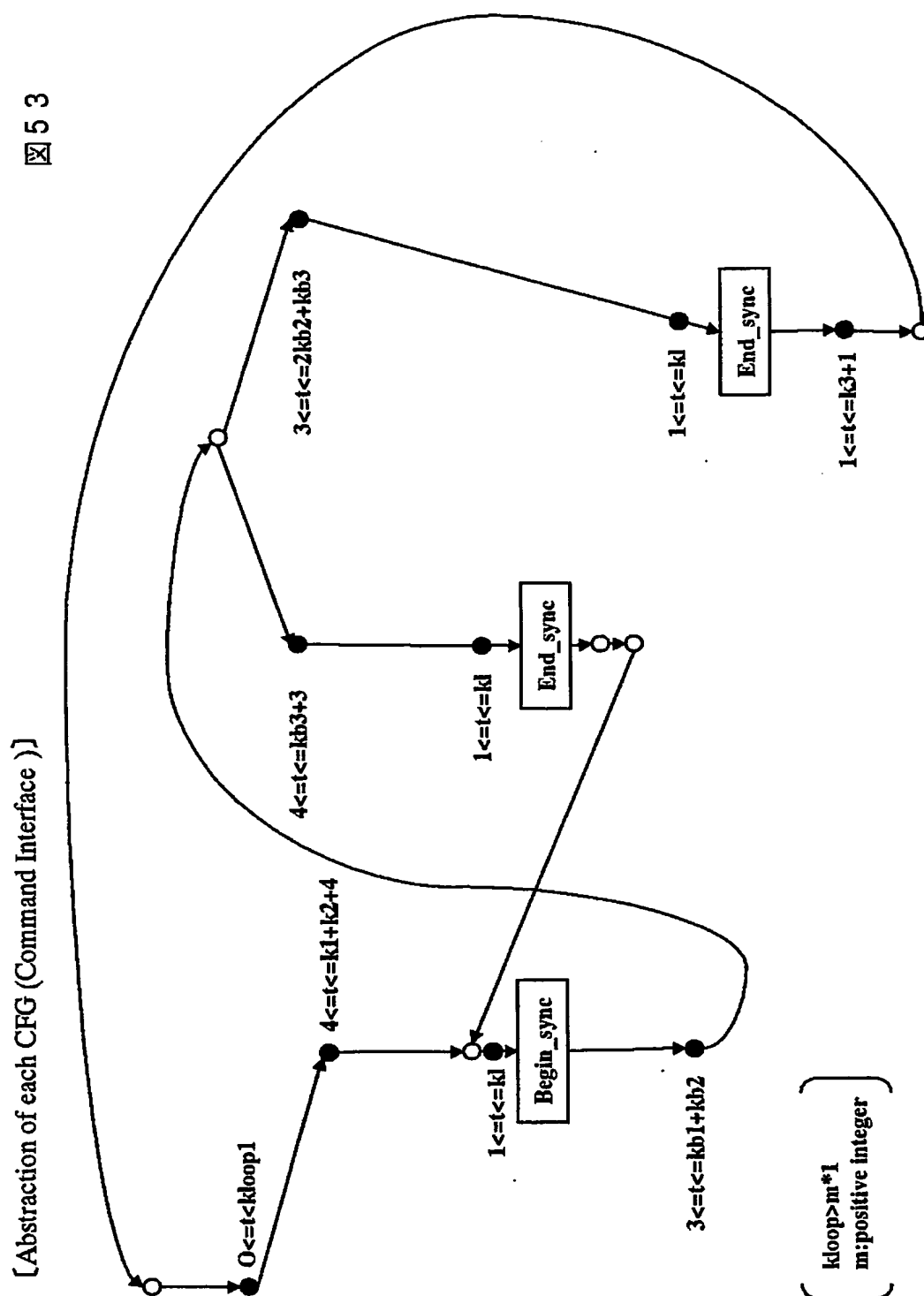
図 5 2

[Abstraction of each CFG (Command Interface)]

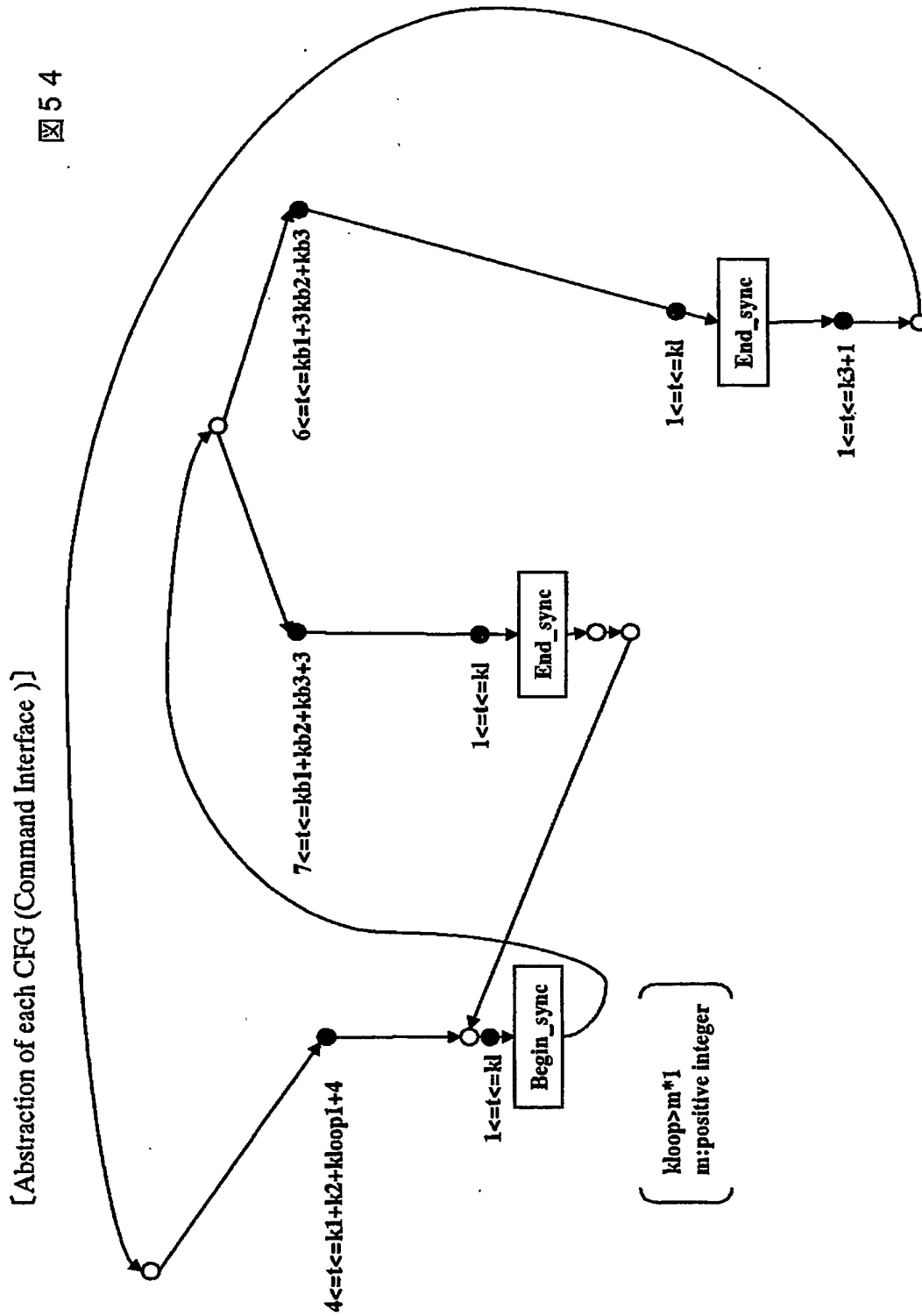


【図 5 3】

3
5
X



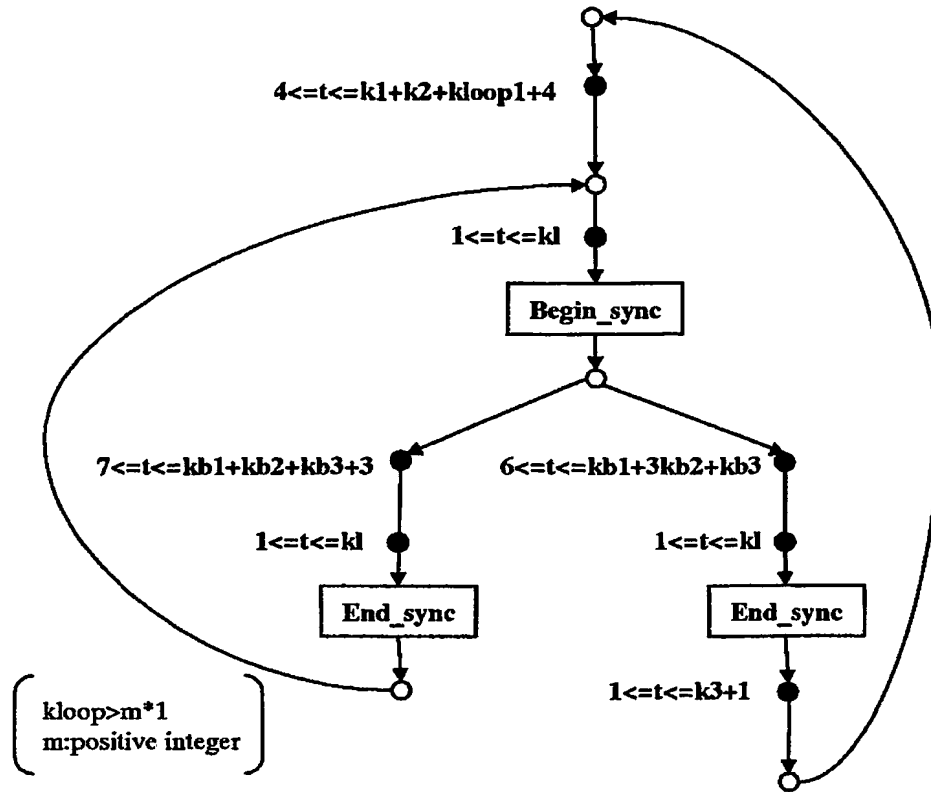
【図 54】



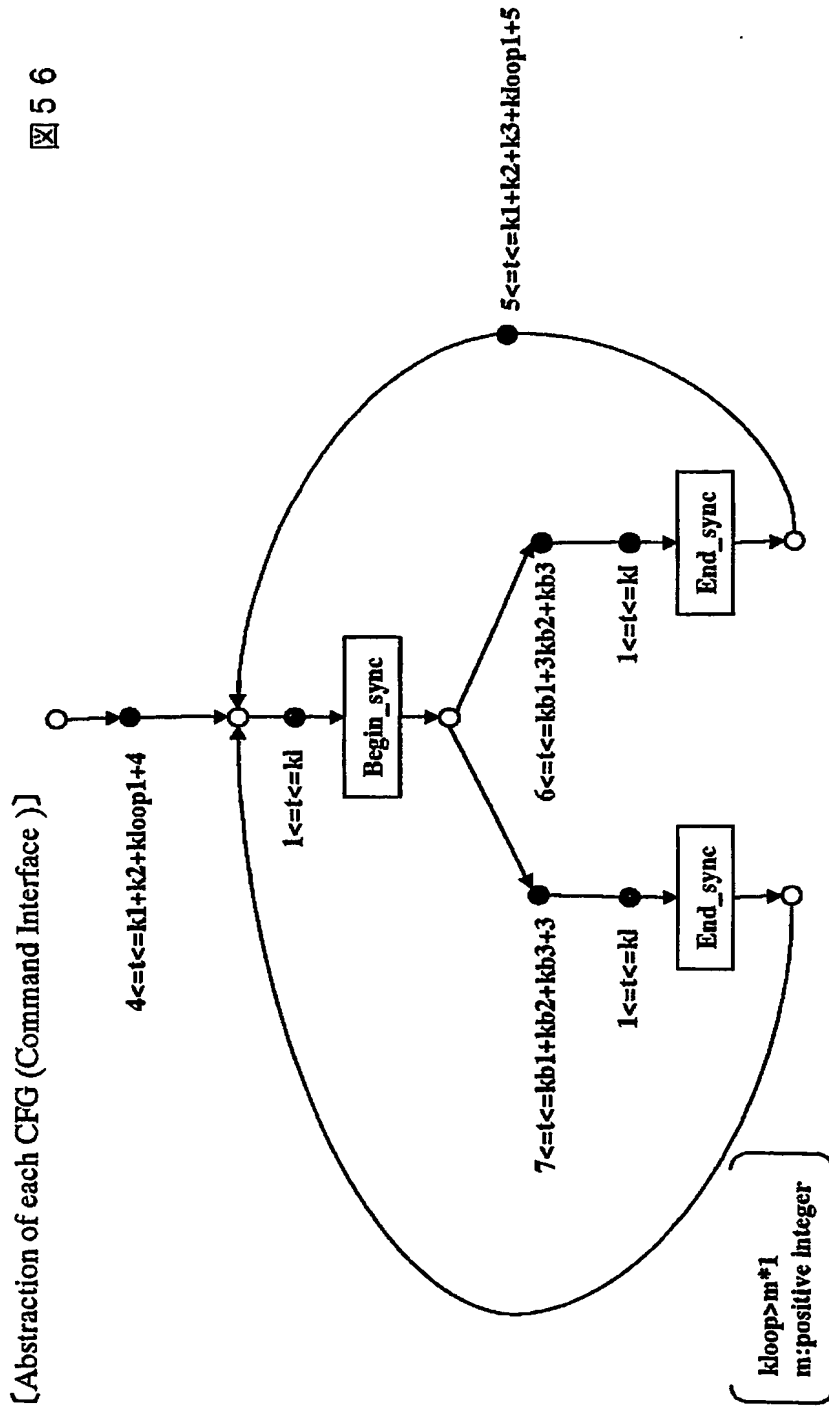
【図 55】

図 55

〔 Abstraction of each CFG (Command Interface) 〕



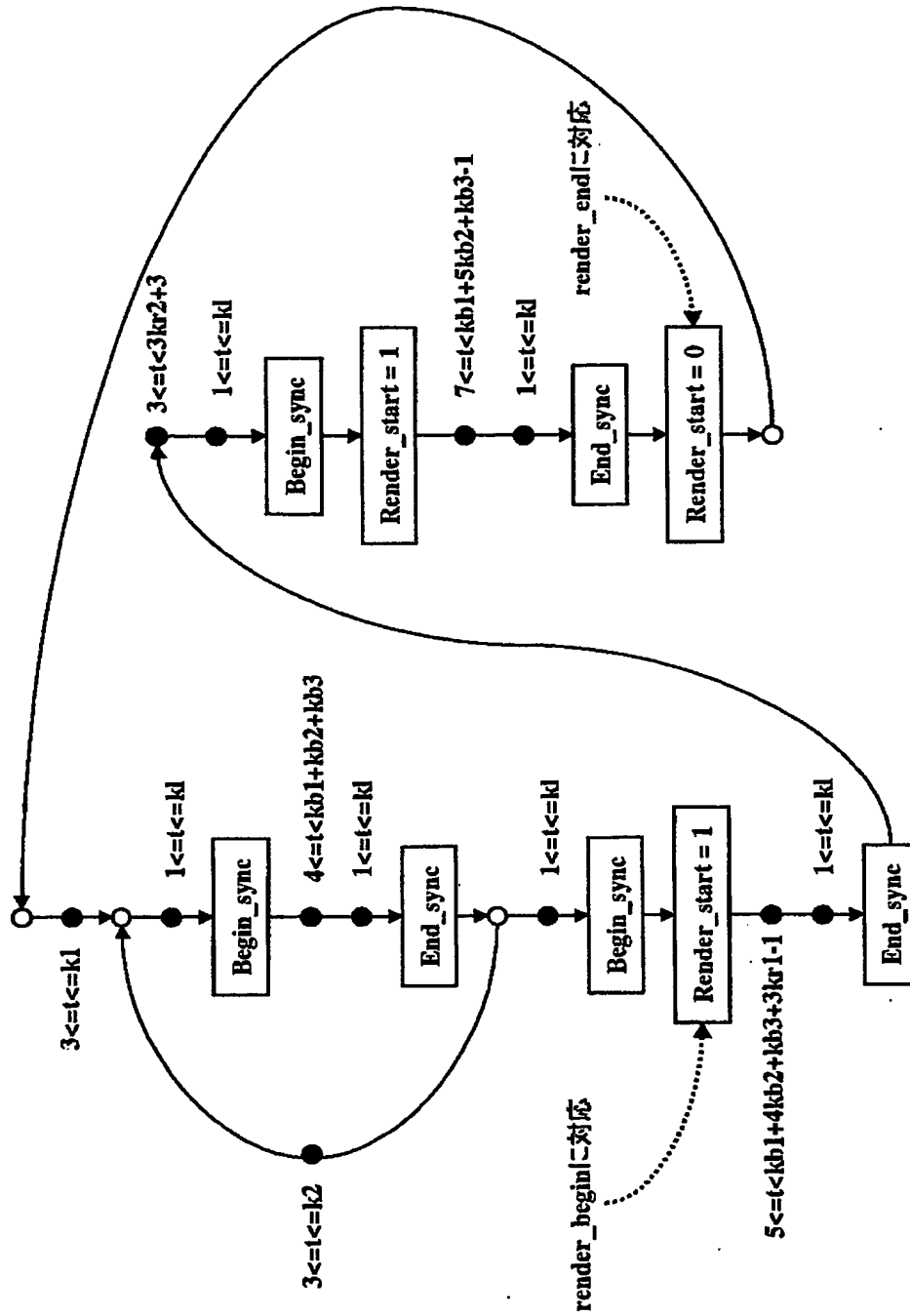
【図 56】



【図 57】

図 57

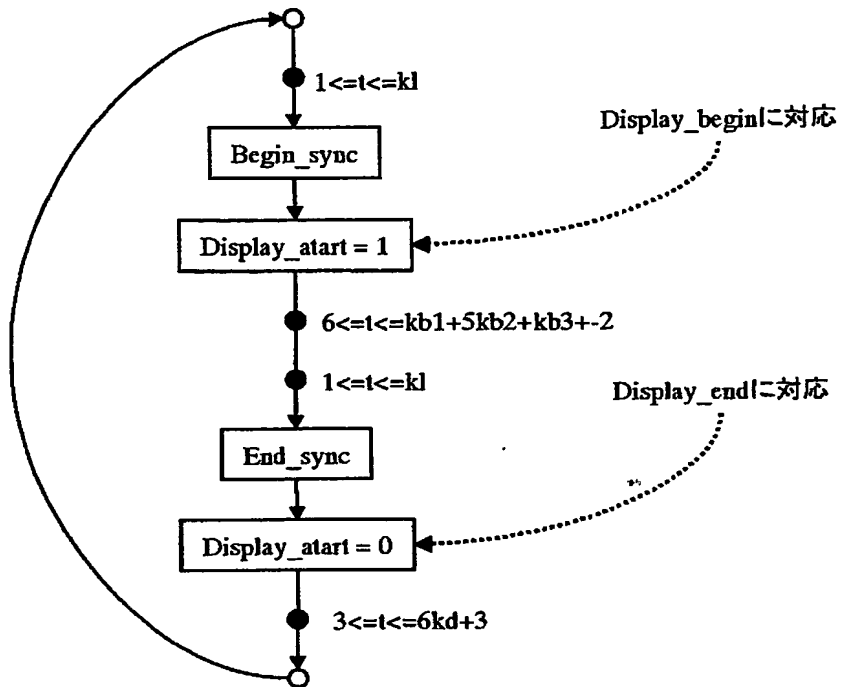
[Abstraction of each CFG (Graphics Rendering Unit)]



【図 58】

図 58

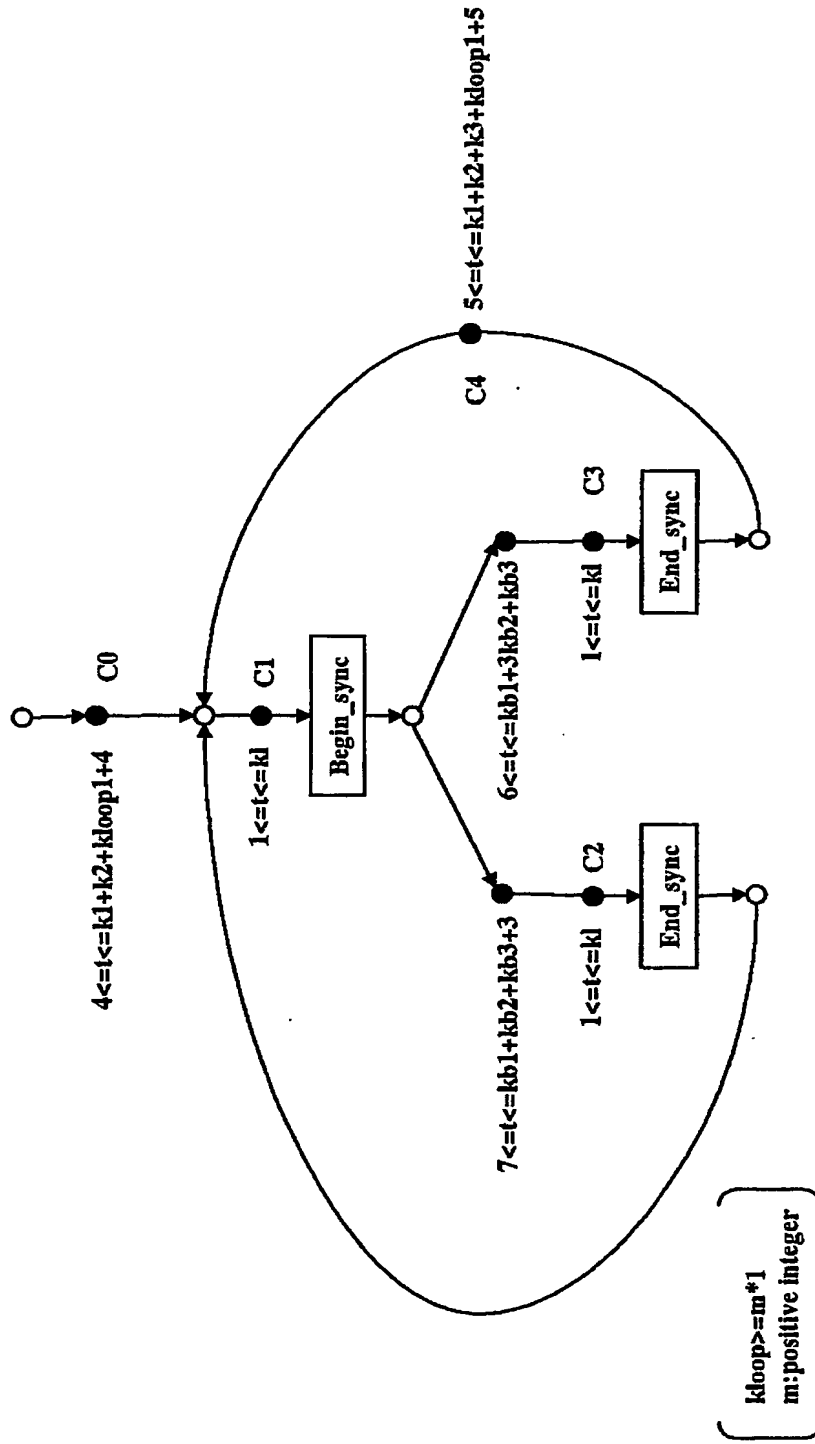
〔 Abstraction of each CFG (Dispaly Unit) 〕



【図 59】

図 59

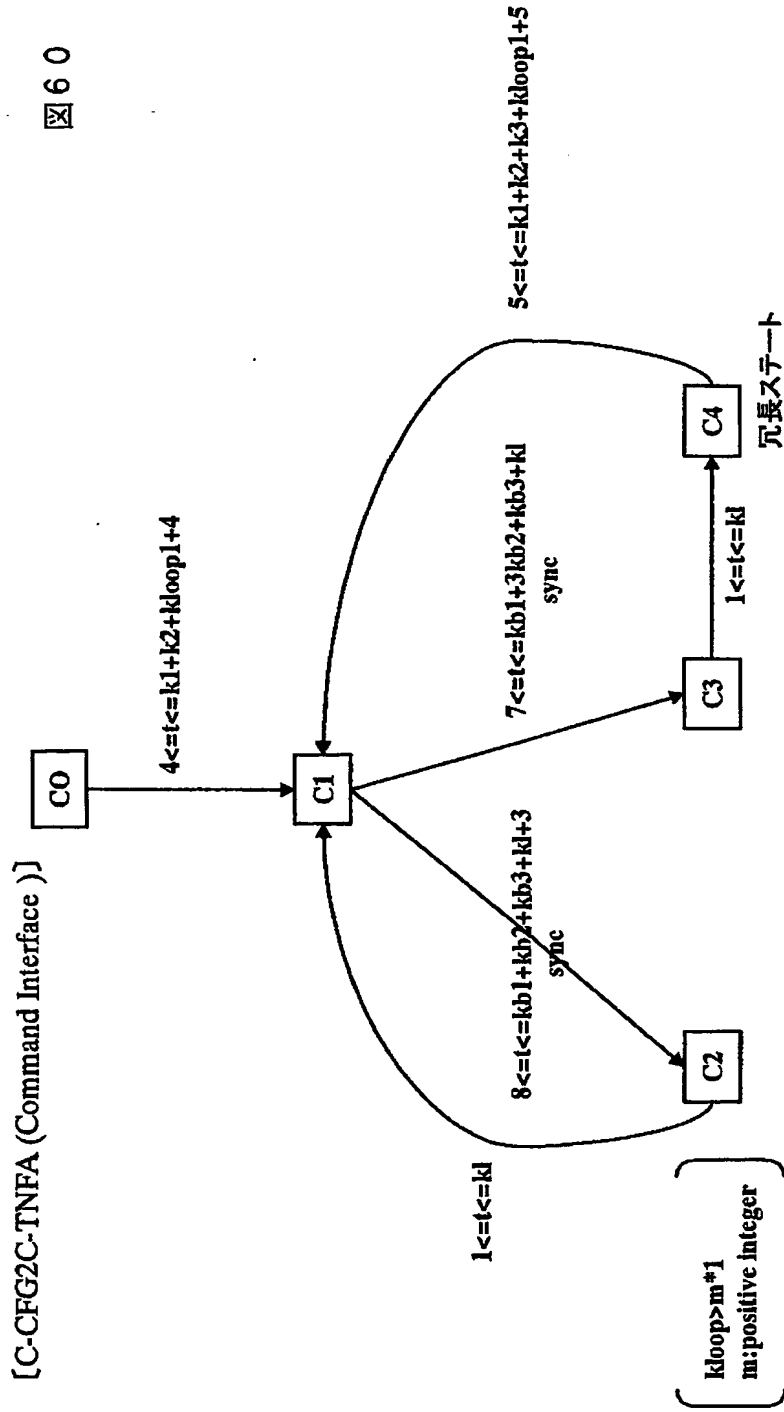
[C-CFG2C-TNFA (Command Interface)]



[C1 => C2, C1 => C3 : Atomic Operation]

【図 60】

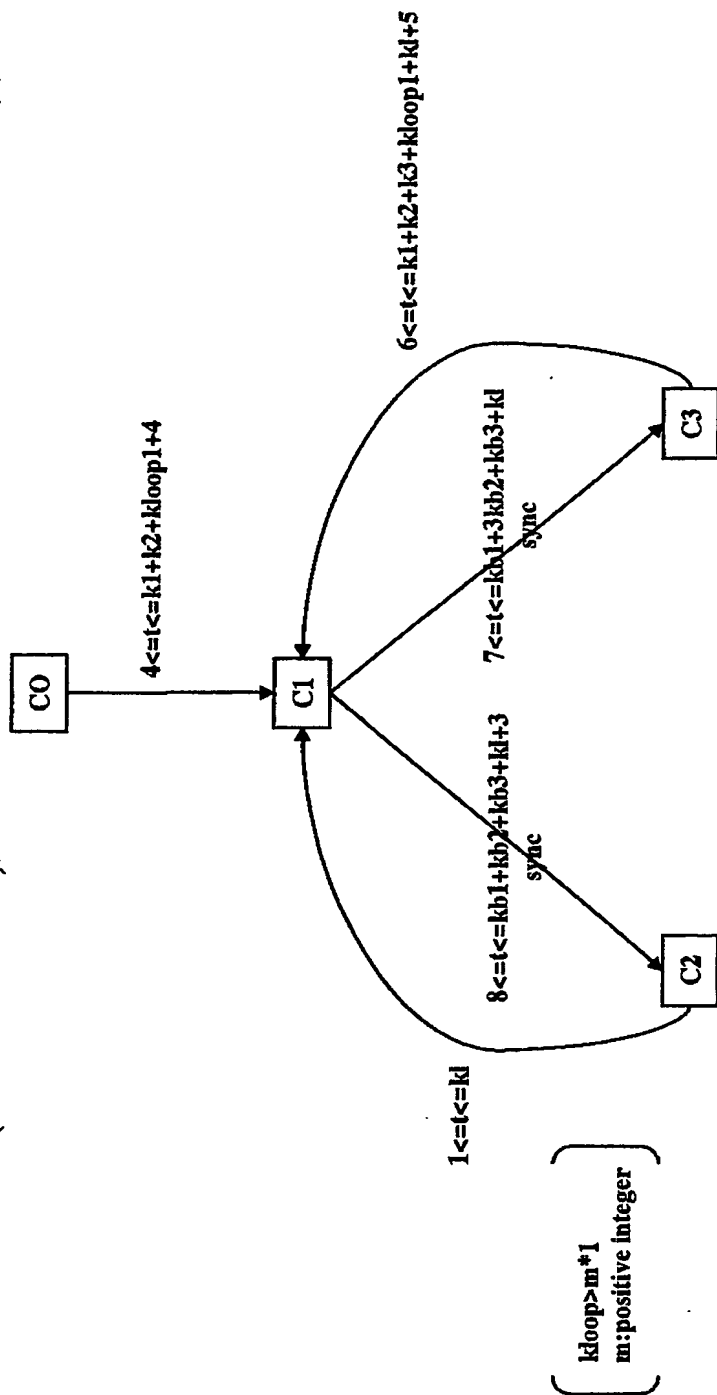
図 60



【図 61】

図 61

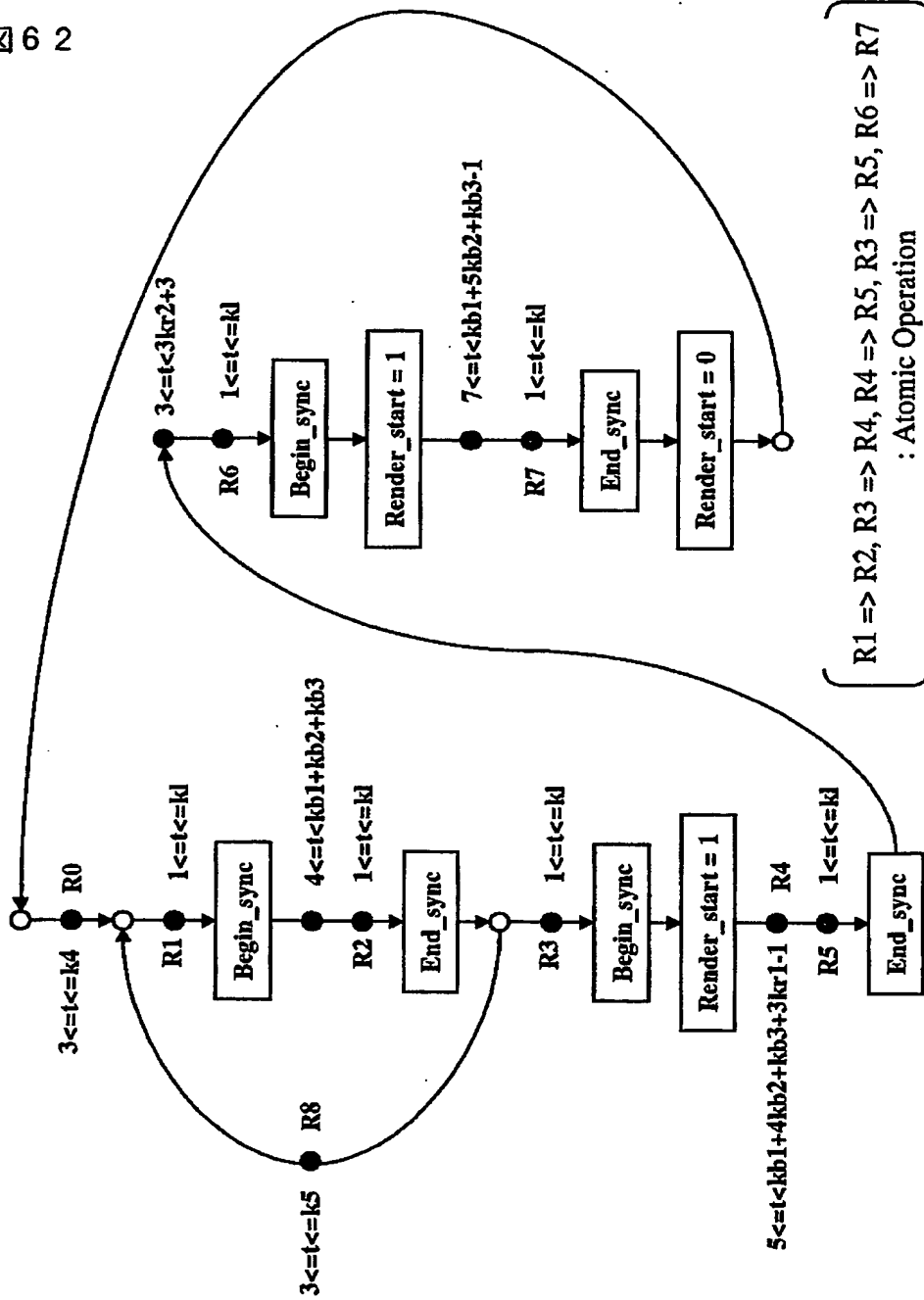
[C-CFG2C-TNFA (Command Interface)]



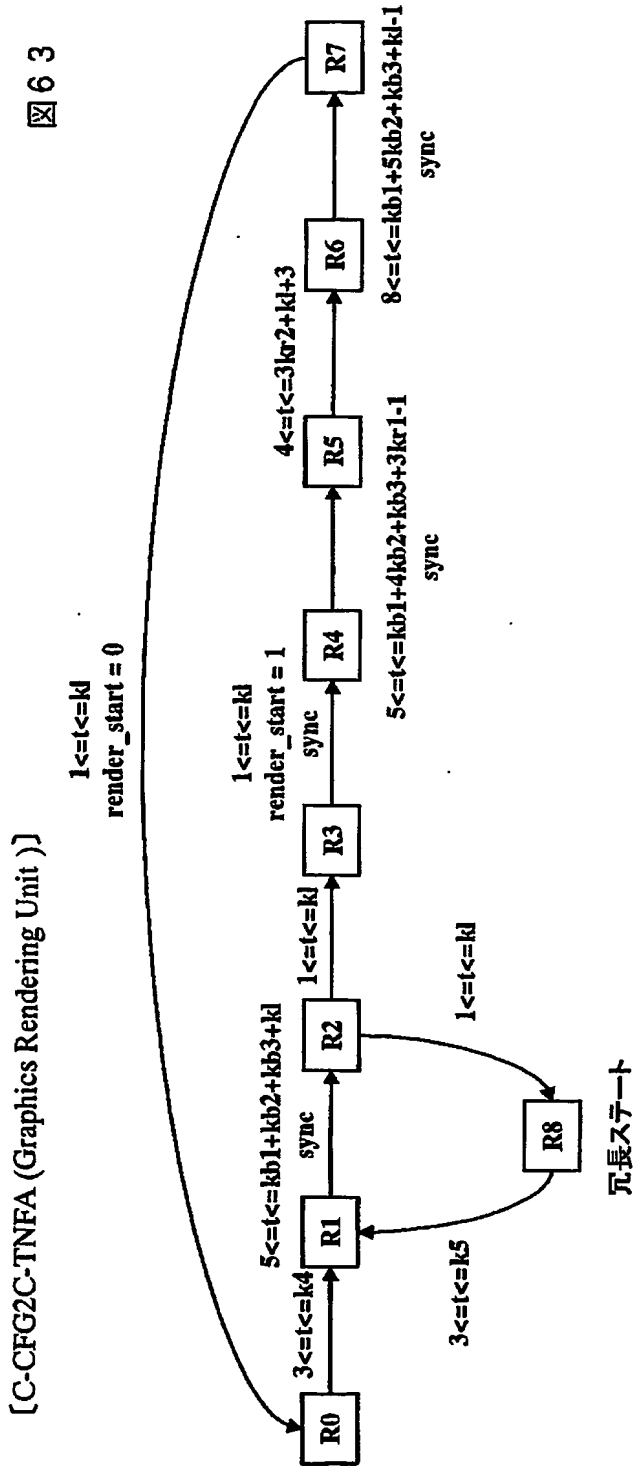
【図 6 2】

図 6 2

[C-CFG2C-TNFA (Graphics Rendering Unit)]



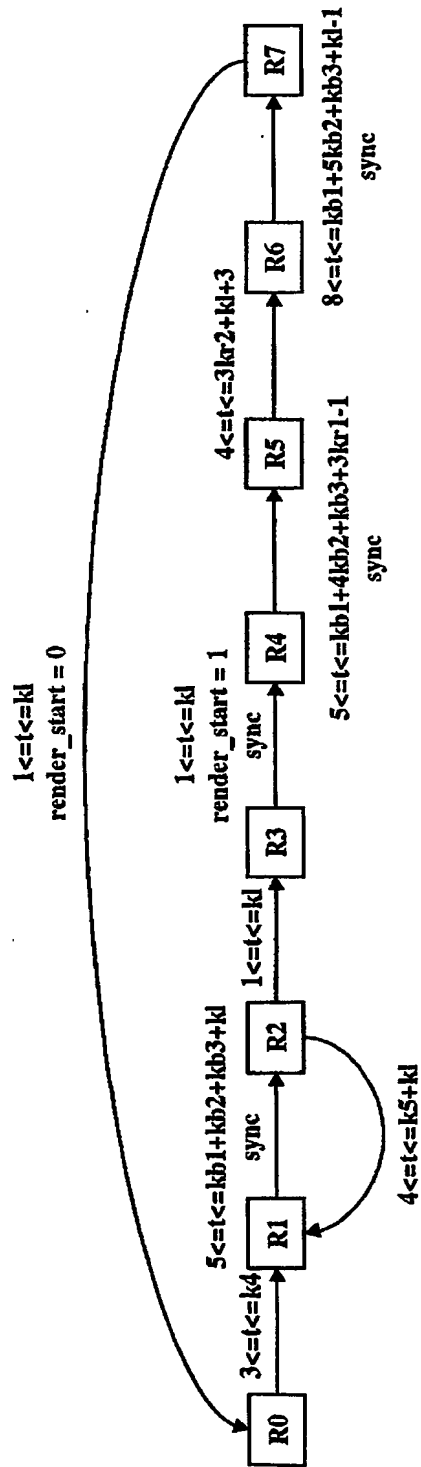
【図 63】



【図 64】

図 64

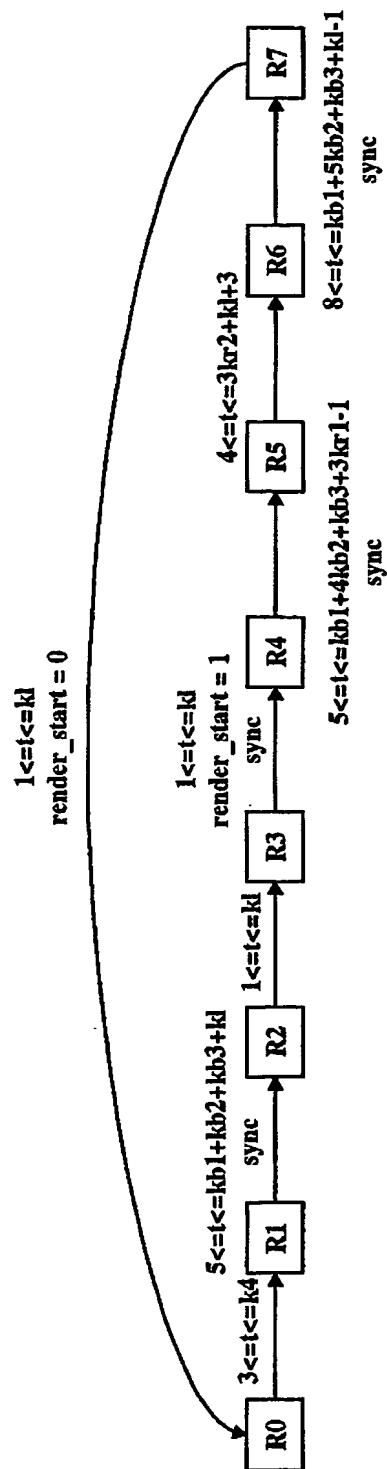
[C-CFG2C-TNFA (Graphics Rendering Unit)]



【図 65】

図 65

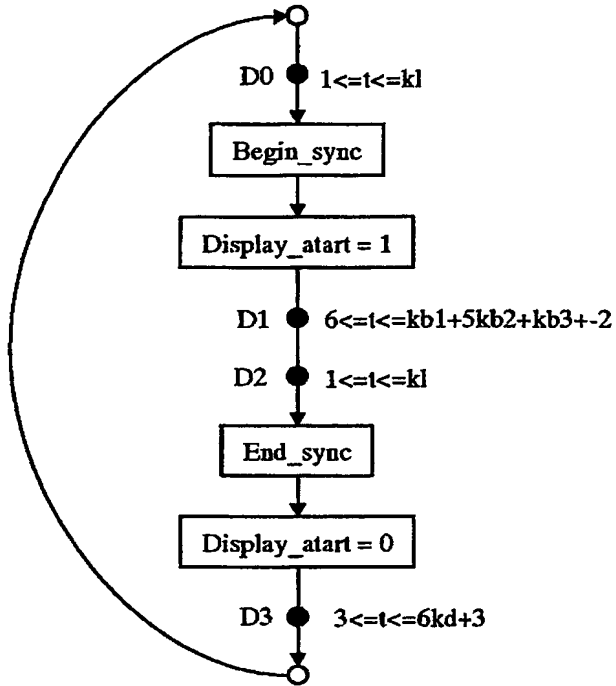
[C-CFG2C-TNFA (Graphics Rendering Unit)]



【図 66】

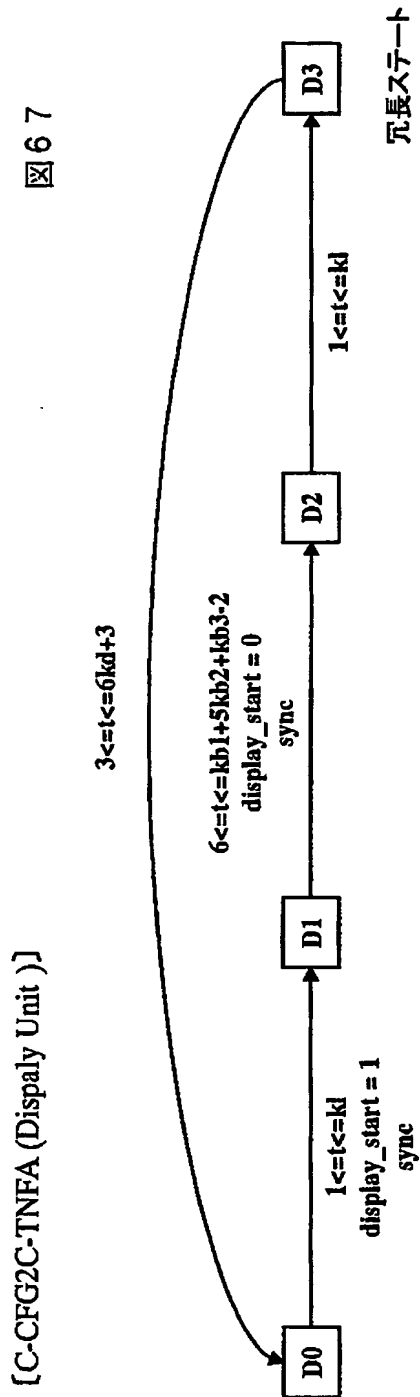
図 66

[C-CFG2C-TNFA(Dispaly Unit)]



[D1 => D1, D1 => D2, D0 => D2 : Atomic Operation]

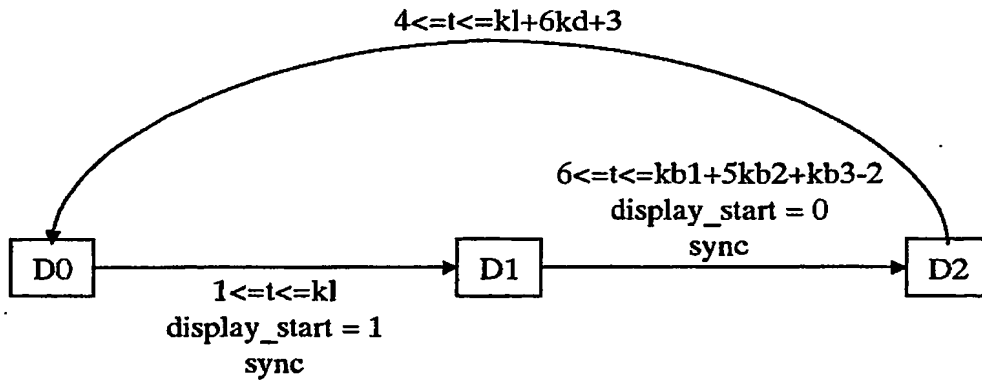
【図 67】



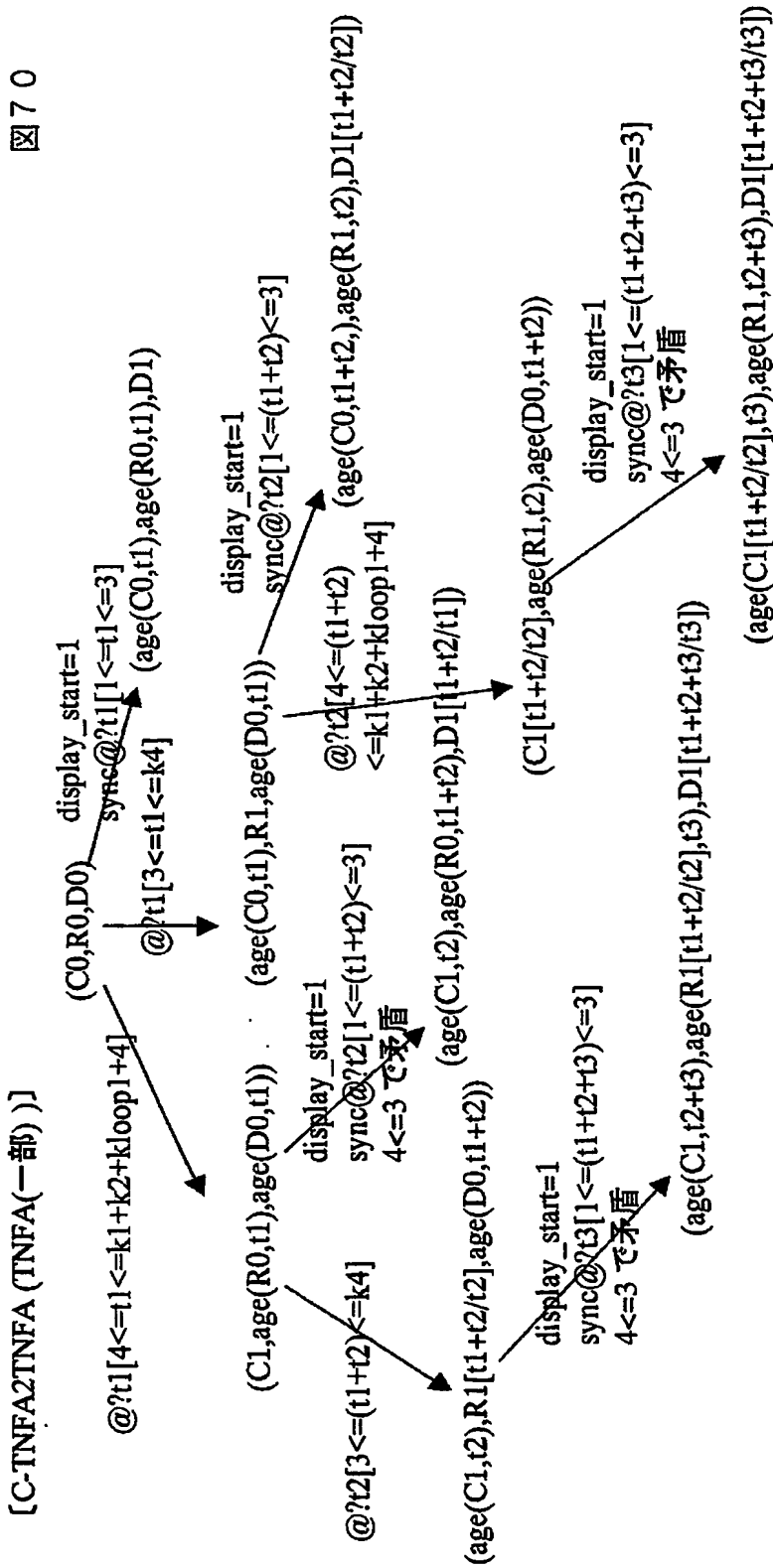
【図 68】

図 68

[C-CFG2C-TNFA(Dispaly Unit)]



【図 70】

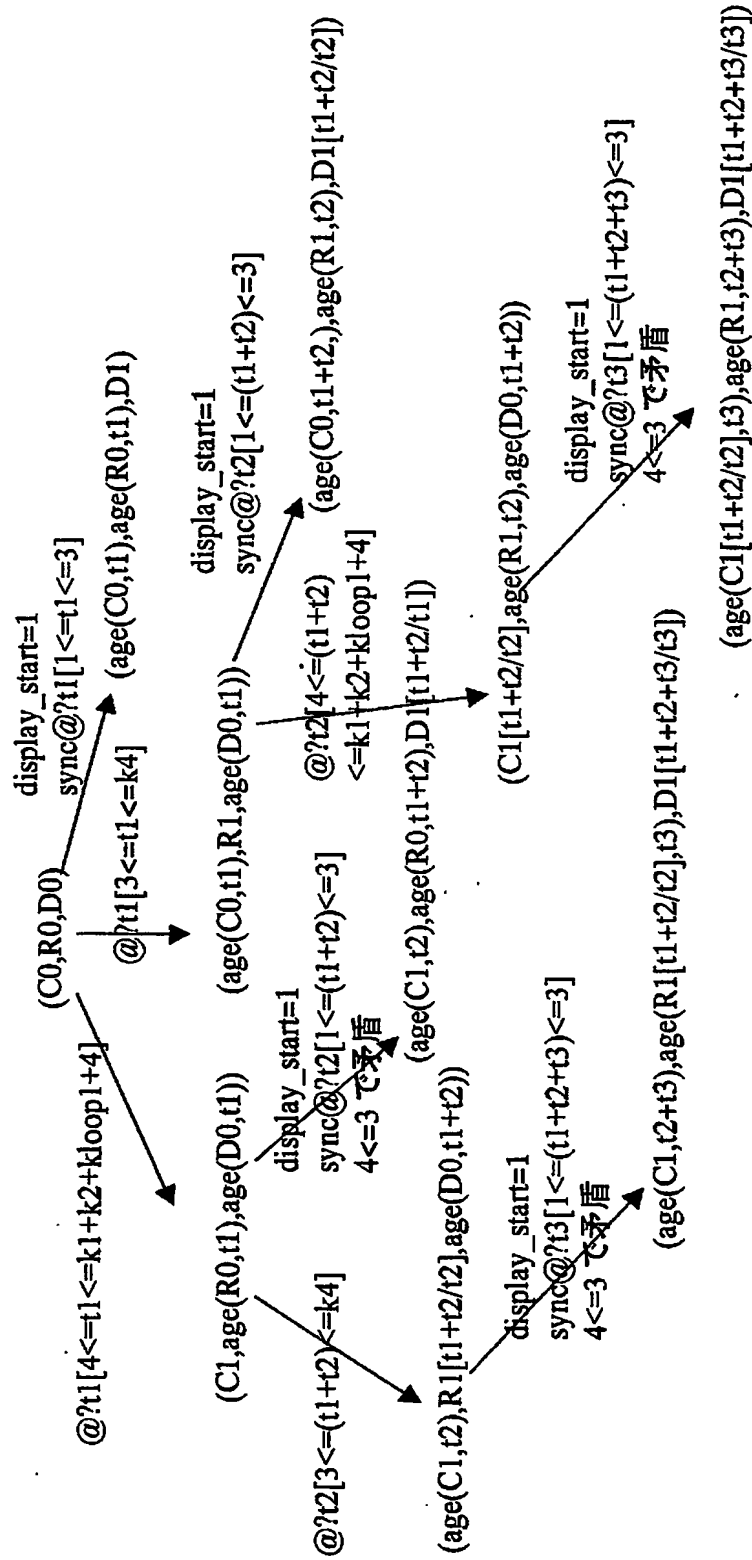


〔仮に、 $k1=3$ だとすると時間制約を満たさない遷移枝が現れる〕

【図 71】

図 71

[C-TNFA2TNFA (TNFA(一部))]]



[時間制約を満たさない遷移枝へのみ到達する状態を削除する]

【図 7 2】

図 7 2

〔 C-TNFA2TNFA(TNFA(一部)) 〕

 $(age(C1[t1+t2/t2], t3), age(R1, t2+t3), D1[t1+t2+t3/t3])$ 

display_start=0
 sync@?t4[6<=t4<=kb1+kb2+kb3-2]

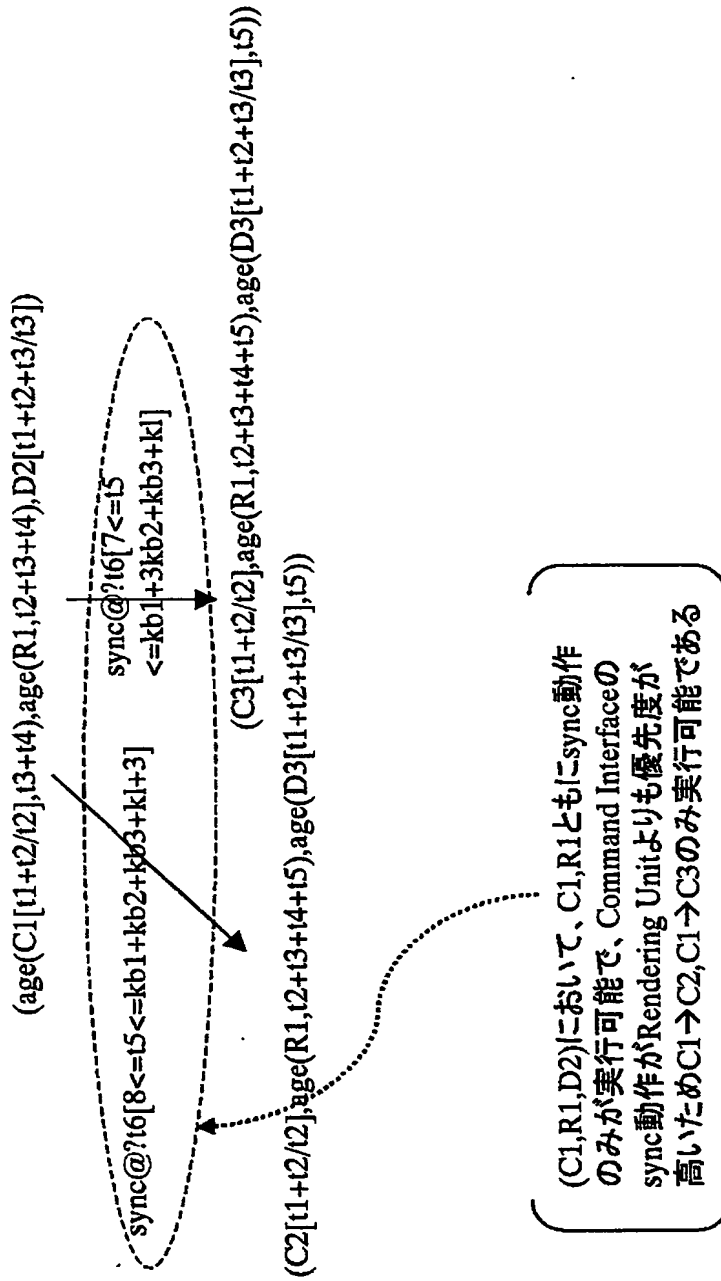
 $(age(C1[t1+t2/t2], t3+t4), age(R1, t2+t3+t4), D2[t1+t2+t3/t3])$

〔 D1→D2の遷移の間、C1,R1はsync動作しか行えず、
 それらはDisplay Unitより優先度が低いため実行されない 〕

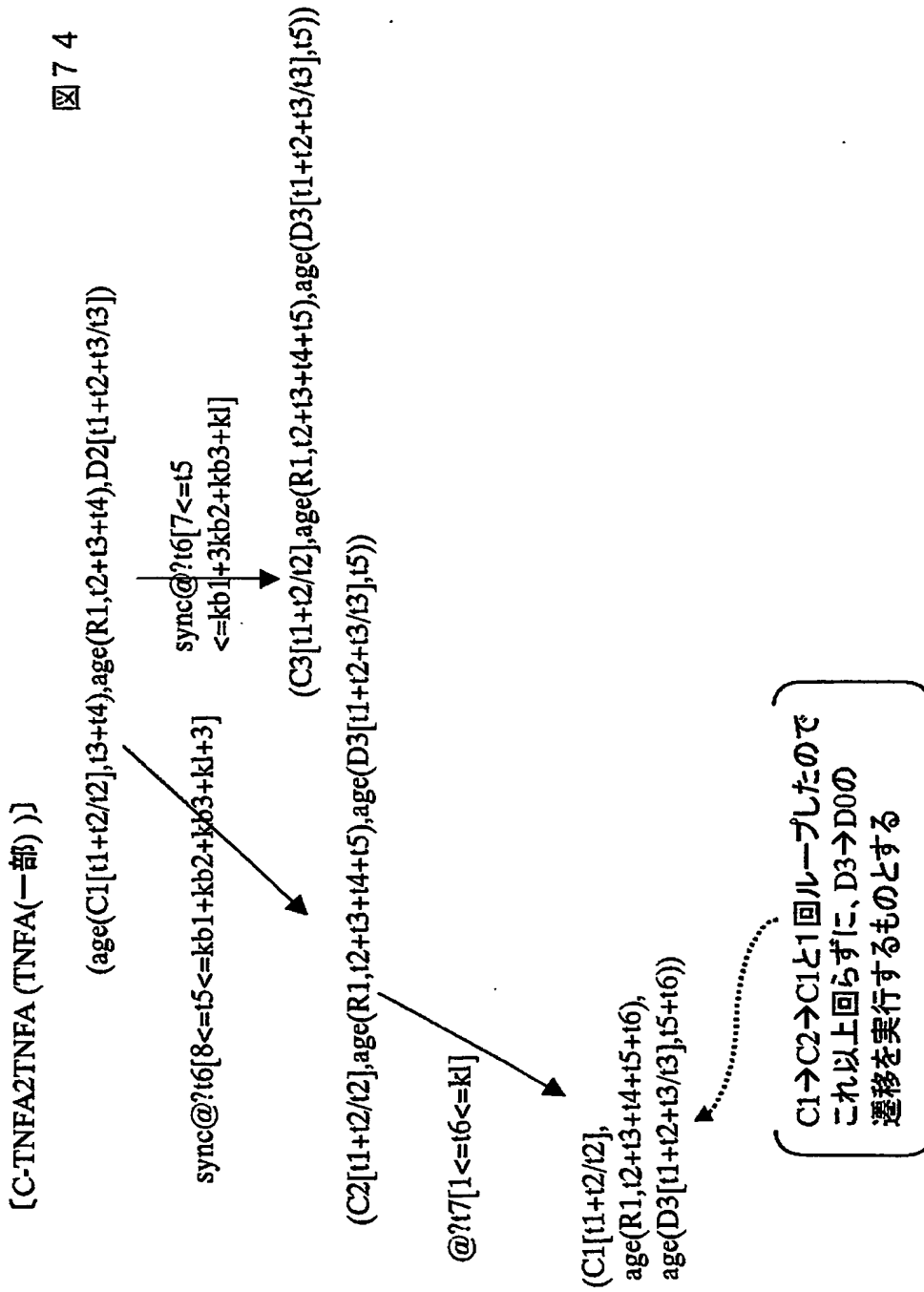
【図 73】

図 73

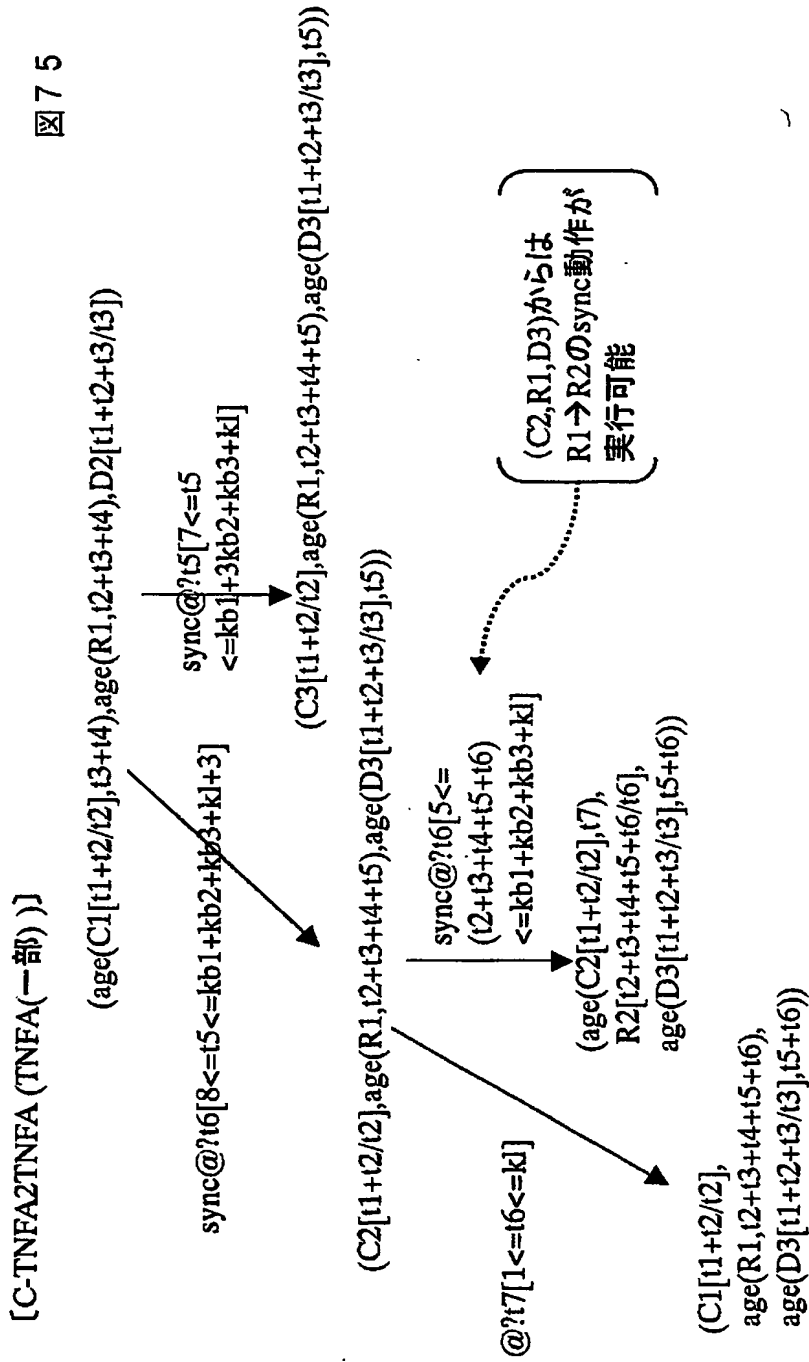
[C-TNFA2TNFA (TNFA(一部))]



【図 74】



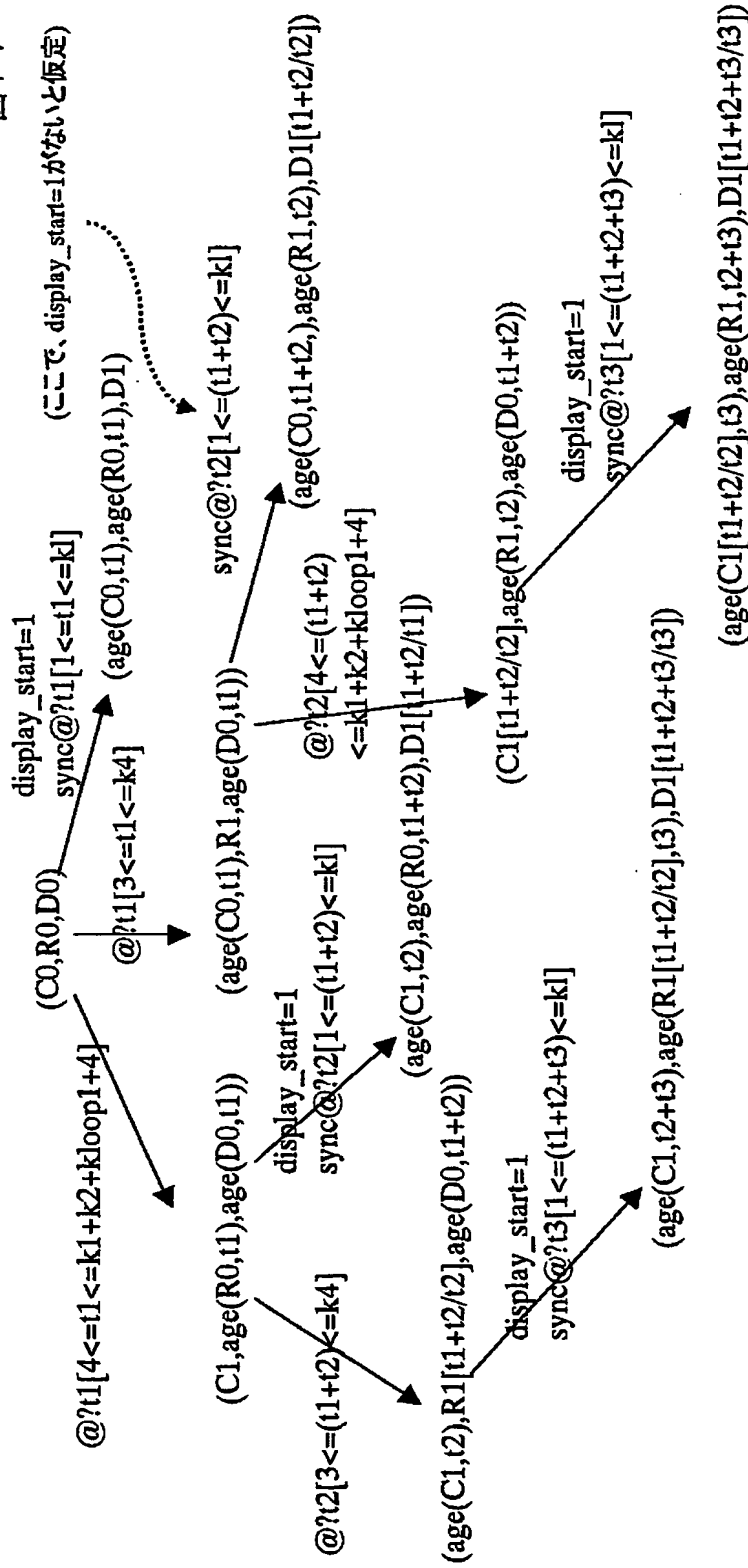
【図 75】



【図 77】

図 77

[Abstraction of TNFA (実行例)]

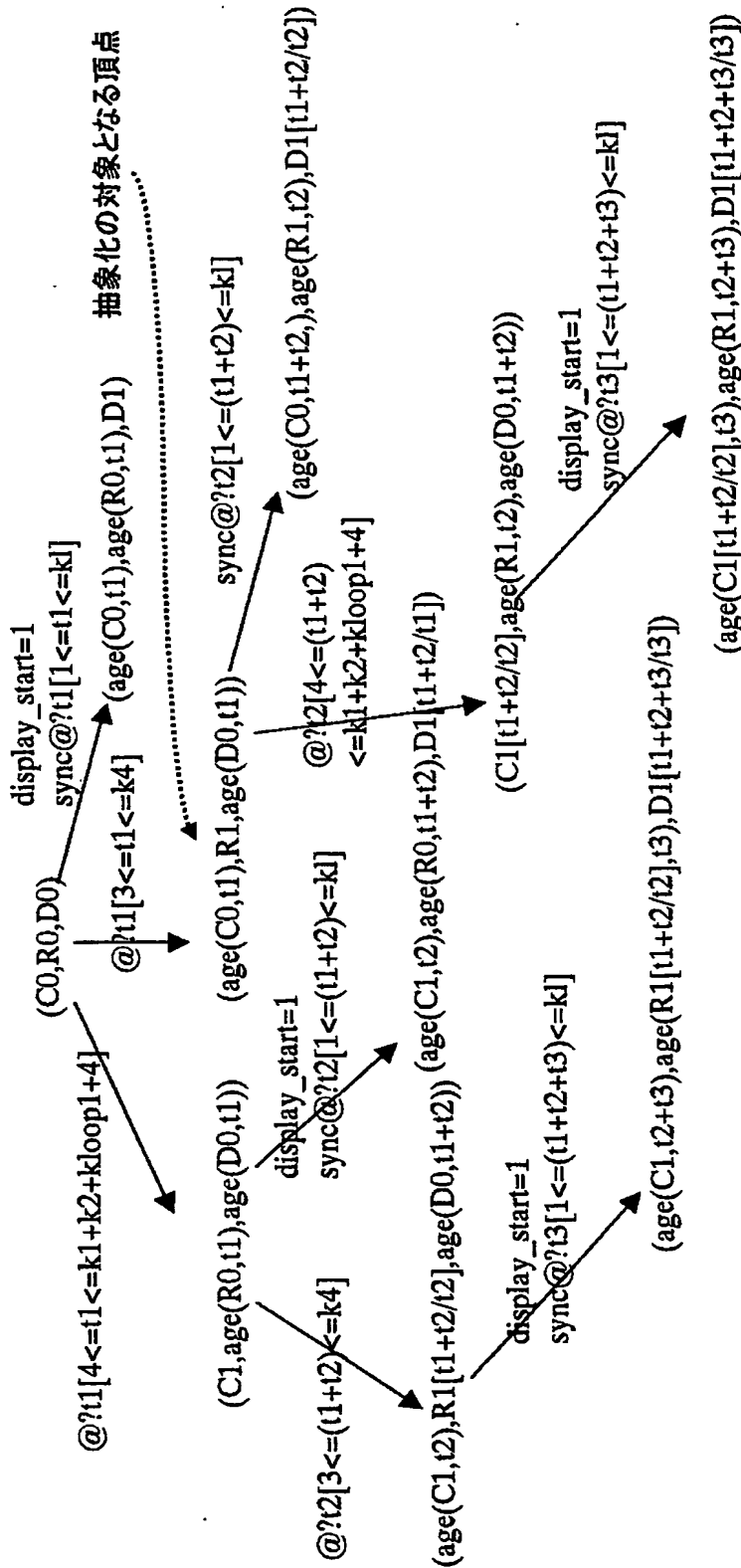


[方針(1)に基づく構成の終了時点]

【図 78】

図 78

[Abstraction of TNFA (実行例)]

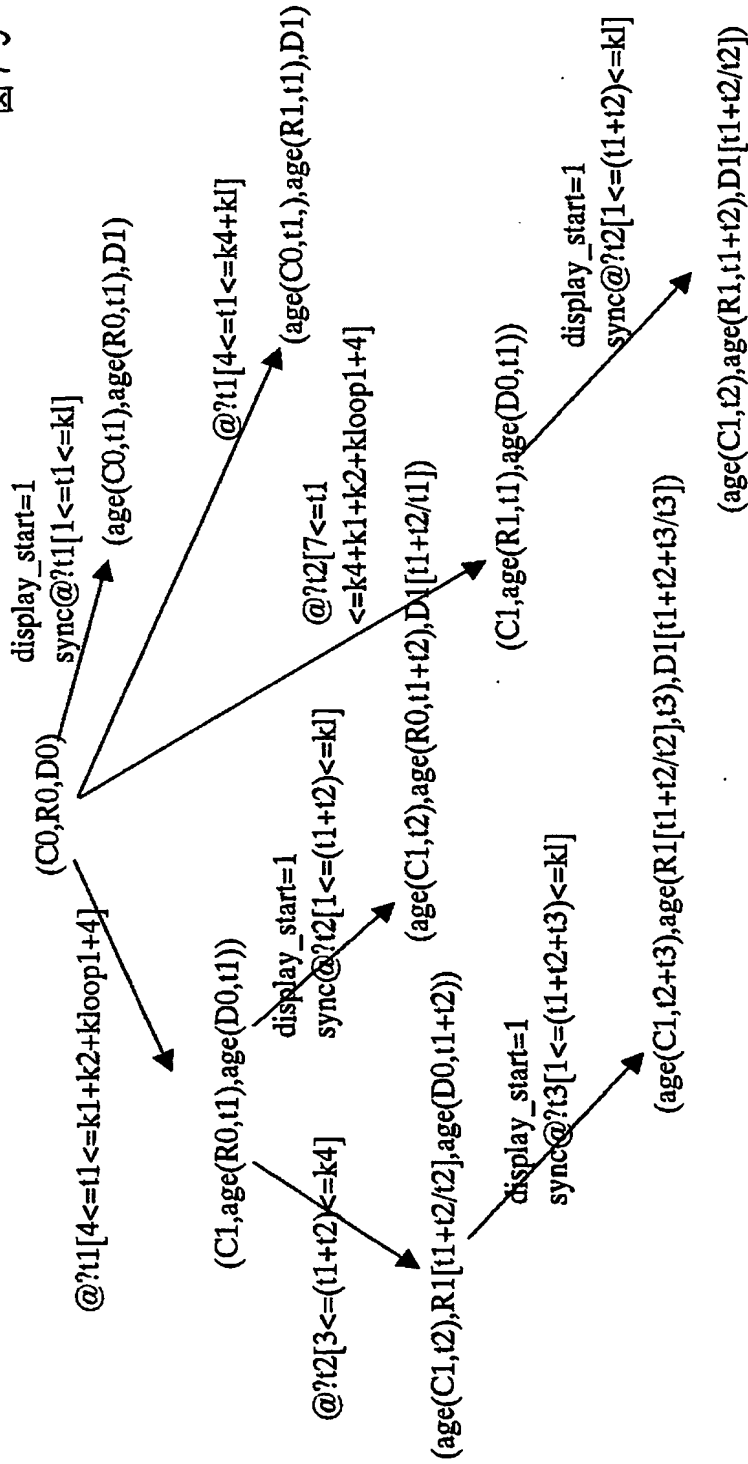


[抽象化対象となる頂点の識別]

【図 79】

図 79

〔Abstraction of TNFA (実行例)〕



〔遷移時間の再計算〕

【図 80】

図 80 [パラメトリック解析結果(実行結果例)]

Value of objective function: 12

k1	0
k2	0
k3	0
k4	4
k5	0
kd	0
kl	4
kloop1	0
kb1	3
kb2	1
kb3	0
kr1	0
kr2	0

【図 81】

図 81 [パラメトリック解析結果(実行結果例)]

Value of objective function: 12

k1	0
k2	0
k3	0
k4	4
k5	0
kd	0
kl	4
kloop1	0
kb1	2
kb2	1
kb3	1
kr1	0
kr2	0

【図 8 2】

図 8 2

[パラメトリック解析結果(実行結果例)]

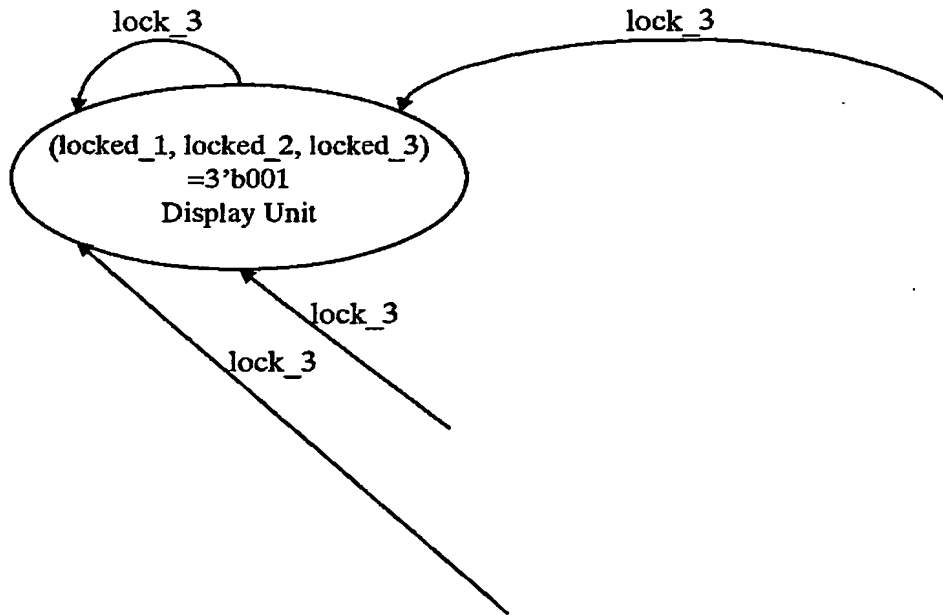
Value of objective function: 19

k1	1
k2	1
k3	1
k4	4
k5	1
kd	1
kl	4
kloop1	1
kb1	2
kb2	1
kb3	1
kr1	0
kr2	1

【図 84】

図 84

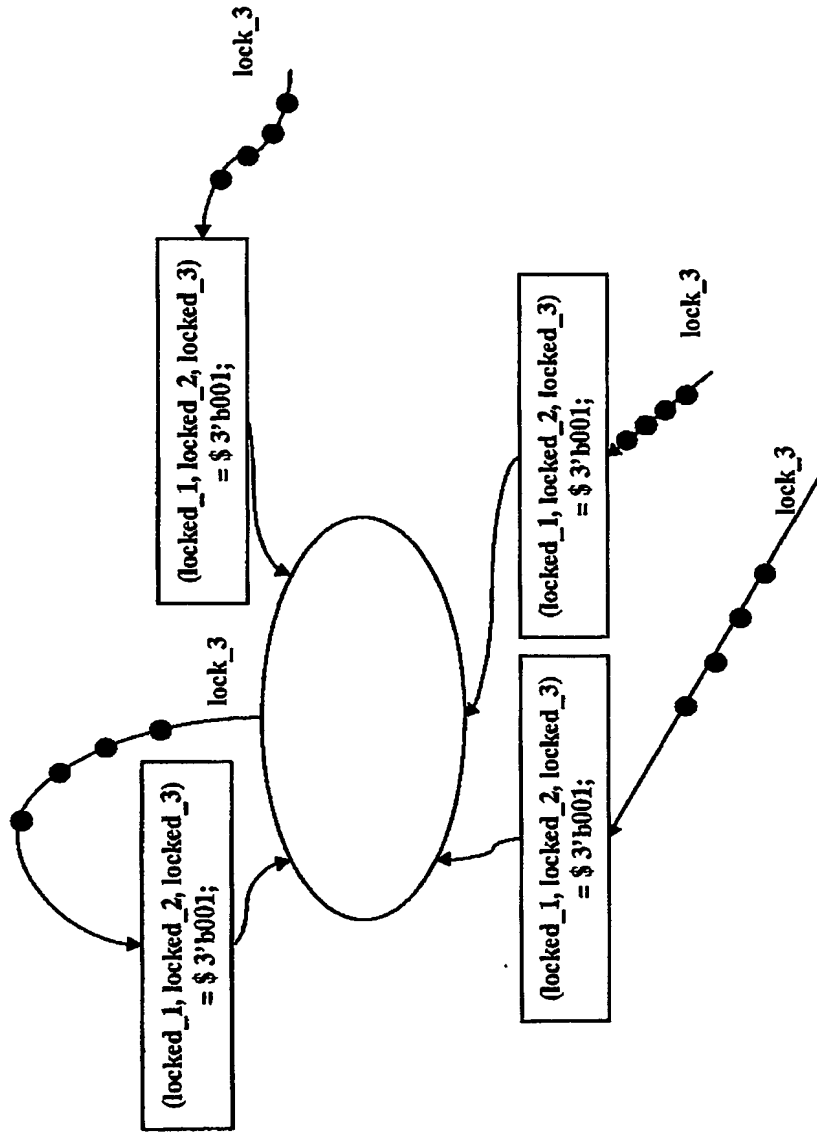
〔ハード合成 (固定優先度スケジューラの修正)〕



【図 85】

図 85

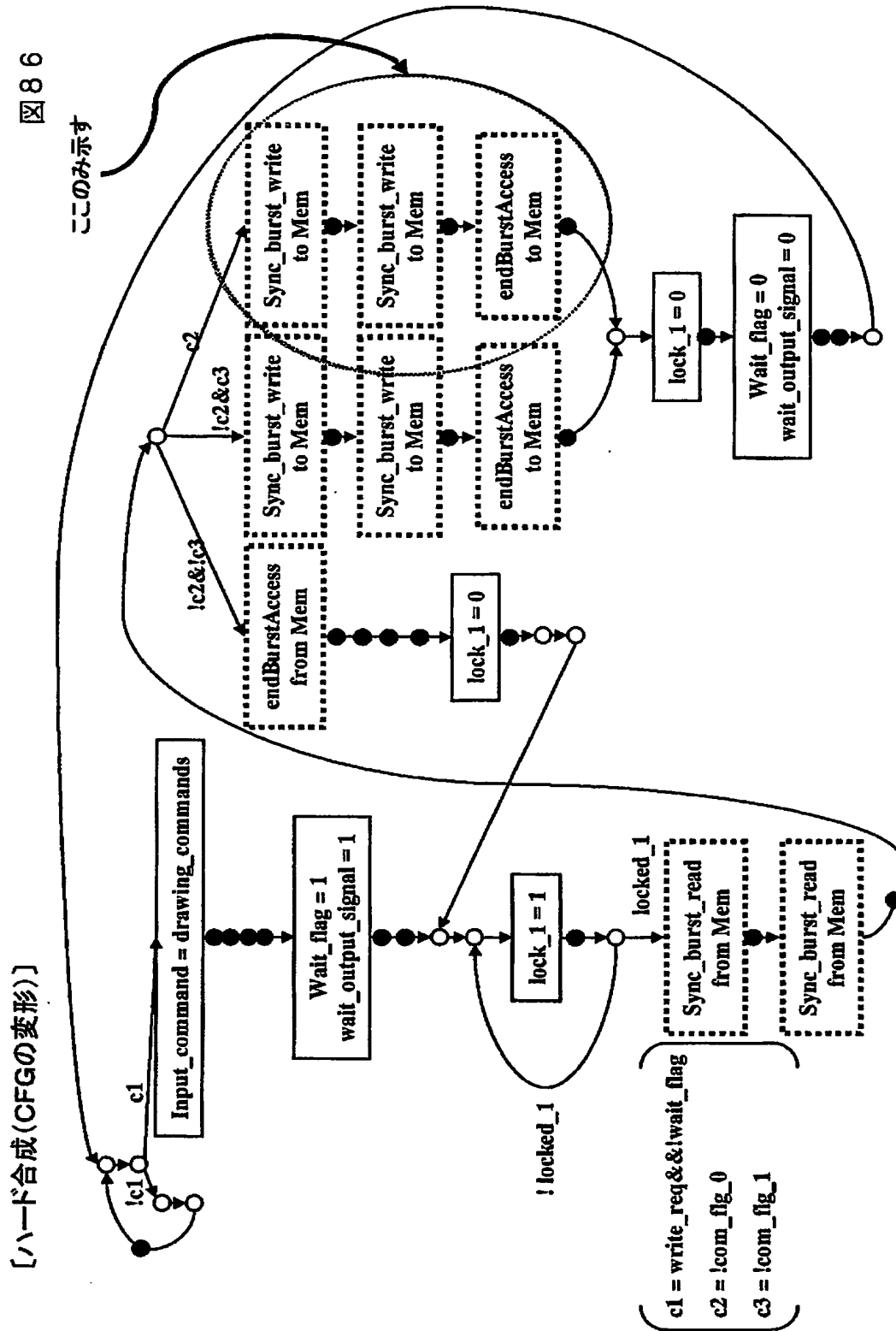
〔ハード合成(固定優先度スケジューラの修正)〕



【図 8 6】

68X

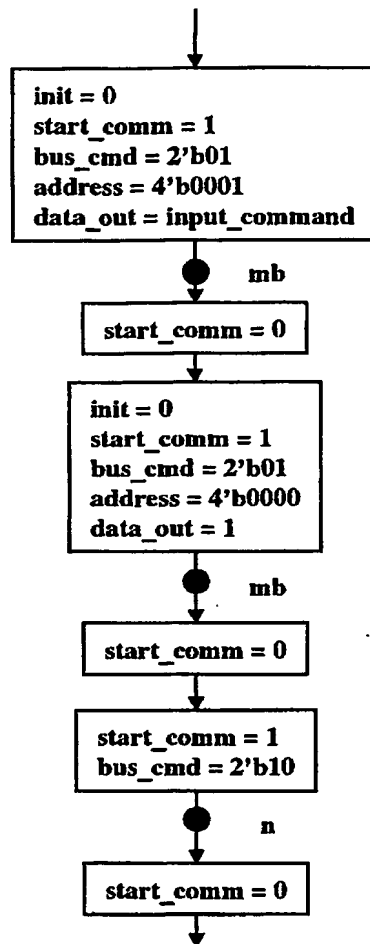
このみす



【図 87】

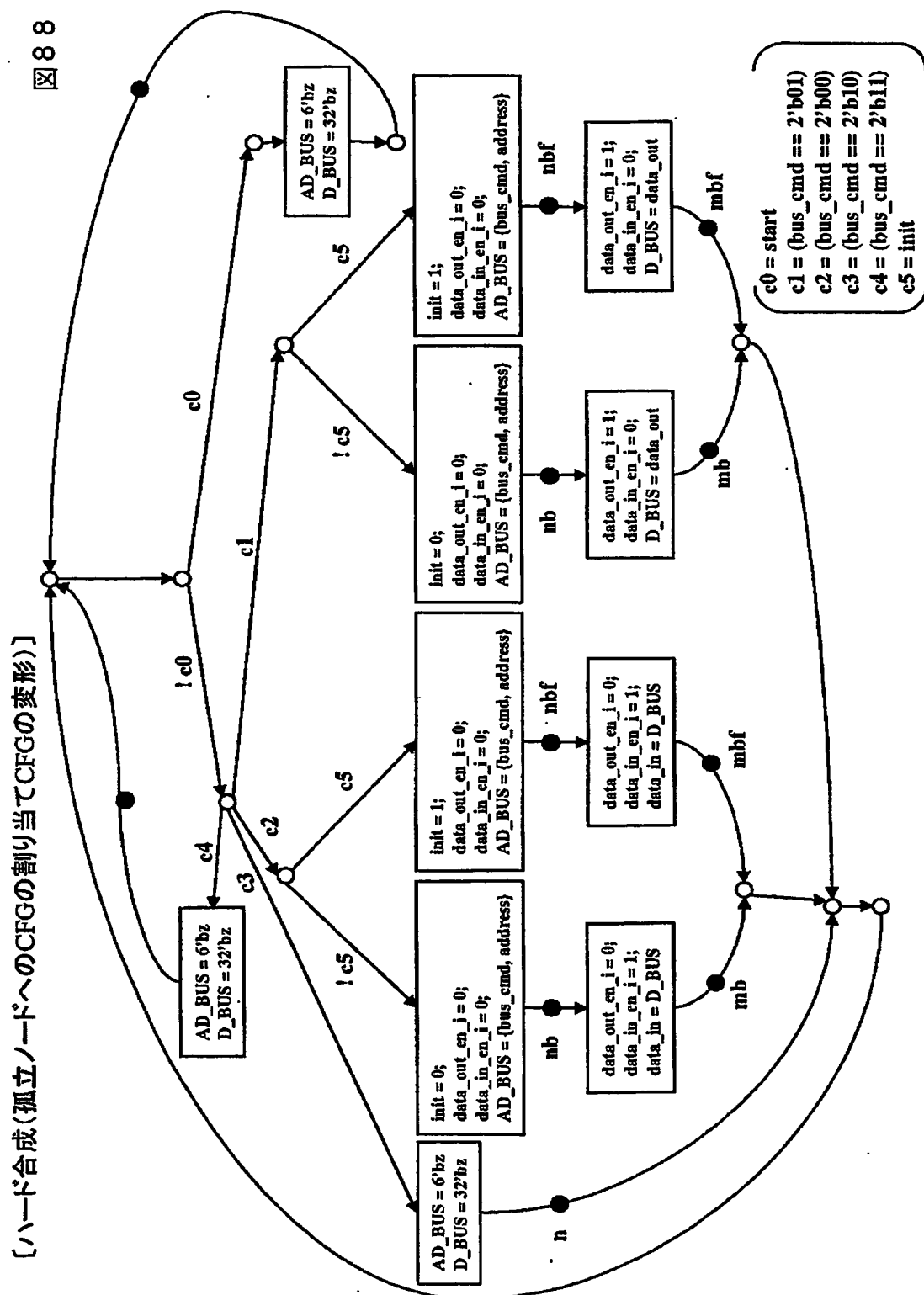
図 87

〔ハード合成 (CFGの変形)〕



【図 8 8】

☐ ☐ ☒



【図 89】

図 89

【ハード合成(共有レジスタ)】

```

while(1) {
    D_BUS = 32'bz;
    if (locked_1 || locked_2 || locked_3) {
        if (AD_BUS[5:4] == 2'b00) {
            if (init) {
                L1:
                if (data_out_en_1 || data_out_en_2 || data_out_en_3) {
                    $
                    register_in();
                } else {
                    $ goto L1;
                }
            } else {
                L2:
                if (data_out_en_1 || data_out_en_2 || data_out_en_3) {
                    register_in();
                } else {
                    $ goto L2;
                }
            }
        }
    }
}

```

```

} else if (AD_BUS == 2'b01) {
    if (init) {
        L3:
        if (data_in_en_1 || data_in_en_2 || data_in_en_3) {
            $
            register_out();
        } else {
            $ goto L3;
        }
    } else {
        L4:
        if (data_in_en_1 || data_in_en_2 || data_in_en_3) {
            register_out();
        } else {
            $ goto L4;
        }
    }
}
}
}
}

```

【図 9 0】

図 9 0
〔ハード合成 (共有レジスタ) 〕

```
void register_in() {  
    if (AD_BUS[3:0] == 4'b0000) {  
        mem_con_reg.current_value[0] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0001) {  
        mem_con_reg.current_value[1] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0010) {  
        mem_con_reg.current_value[2] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0011) {  
        mem_con_reg.current_value[3] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0100) {  
        mem_con_reg.current_value[4] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0101) {  
        mem_con_reg.current_value[5] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0110) {  
        mem_con_reg.current_value[6] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b0111) {  
        mem_con_reg.current_value[7] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b1000) {  
        mem_con_reg.current_value[8] = $ D_BUS;  
    } else if (AD_BUS[3:0] == 4'b1001) {  
        mem_con_reg.current_value[9] = $ D_BUS;  
    }  
}
```

【図 9 1】

図 9 1 [ハード合成 (共有レジスタ)]

```
void register_out() {  
  if (AD_BUS[3:0] == 4'b0000) {  
    D_BUS = mem_con_reg.current_value[0];  
  } else if (AD_BUS[3:0] == 4'b0001) {  
    D_BUS = mem_con_reg.current_value[1];  
  } else if (AD_BUS[3:0] == 4'b0010) {  
    D_BUS = mem_con_reg.current_value[2];  
  } else if (AD_BUS[3:0] == 4'b0011) {  
    D_BUS = mem_con_reg.current_value[3];  
  } else if (AD_BUS[3:0] == 4'b0100) {  
    D_BUS = mem_con_reg.current_value[4];  
  } else if (AD_BUS[3:0] == 4'b0101) {  
    D_BUS = mem_con_reg.current_value[5];  
  } else if (AD_BUS[3:0] == 4'b0110) {  
    D_BUS = mem_con_reg.current_value[6];  
  } else if (AD_BUS[3:0] == 4'b0111) {  
    D_BUS = mem_con_reg.current_value[7];  
  } else if (AD_BUS[3:0] == 4'b1000) {  
    D_BUS = mem_con_reg.current_value[8];  
  } else if (AD_BUS[3:0] == 4'b1001) {  
    D_BUS = mem_con_reg.current_value[9];  
  }  
}
```

【書類名】 要約書

【要約】

【課題】 並列動作を記述可能なプログラム言語とパラメトリック・モデルチェックキングを用いた実時間制約を満たすバス・システムの新規設計手法を提供する。

【解決手段】 並列動作の記述が可能なプログラム言語を用いて複数のデバイスを定義したプログラム記述（１）を入力し、入力したプログラム記述を中間表現に変換し（Ｓ２）、中間表現に対し、実時間制約を満足するパラメータを生成し（Ｓ３）、生成したパラメータに基づいてハードウェア記述言語による回路記述を合成する（Ｓ４）。中間表現は、コンカレントなコントロールフローフラグ、コンカレントなパラメータ付き時間オートマトン等である。前記パラメータ生成に、パラメトリック・モデルチェックキングを行う。プログラム記述は、runメソッドを用いてデバイスの定義を行い、バリア同期を用いてデバイスのクロック同期を定義する。

【選択図】 図１

【書類名】 出願人名義変更届（一般承継）

【あて先】 特許庁長官 殿

【事件の表示】

【出願番号】 特願2002-313201

【承継人】

【識別番号】 503121103

【氏名又は名称】 株式会社ルネサステクノロジ

【承継人代理人】

【識別番号】 100089071

【弁理士】

【氏名又は名称】 玉村 静世

【提出物件の目録】

【包括委任状番号】 0308734

【物件名】 承継人であることを証明する登記簿謄本 1

【援用の表示】 特許第3154542号 平成15年4月11日付け
提出の会社分割による特許権移転登録申請書 を援用
する

【物件名】 権利の承継を証明する承継証明書 1

【援用の表示】 特願平2-321649号 同日提出の出願人
名義変更届（一般承継）を援用する

【プルーフの要否】 要

認定・付加情報

特許出願の番号	特願 2002-313201
受付番号	50301210814
書類名	出願人名義変更届 (一般承継)
担当官	土井 恵子 4264
作成日	平成15年10月 7日

<認定情報・付加情報>

【提出日】 平成15年 7月23日

特願 2 0 0 2 - 3 1 3 2 0 1

出 願 人 履 歷 情 報

識別番号

[0 0 0 0 0 5 1 0 8]

1. 変更年月日

1 9 9 0 年 8 月 3 1 日

[変更理由]

新規登録

住 所

東京都千代田区神田駿河台 4 丁目 6 番地

氏 名

株式会社日立製作所

特願 2002-313201

出願人履歴情報

識別番号

[503121103]

1. 変更年月日

2003年 4月 1日

[変更理由]

新規登録

住 所

東京都千代田区丸の内二丁目4番1号

氏 名

株式会社ルネサステクノロジ